РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

С. М. Окулов

ПРОГРАММИРОВАНИЕ В АЛГОРИТМАХ

ИЗДАТЕЛЬСТВО БИНОМ

РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

С. М. Окулов

ПРОГРАММИРОВАНИЕ В АЛГОРИТМАХ

5-е издание (электронное)

Москва
БИНОМ. Лаборатория знаний
2014

УДК 519.85(023)
ББК 22.18
О-52

Серия основана в 2008 г.

Окулов С. М.

О-52 Программирование в алгоритмах [Электронный ресурс] / С. М. Окулов. — 5-е изд. (эл.). — М. : БИНОМ. Лаборатория знаний, 2014. — 383 с. : ил. — (Развитие интеллекта школьников).

ISBN 978-5-9963-2311-1

Искусство программирования представлено в виде учебного курса, раскрывающего секреты наиболее популярных алгоритмов. Освещены такие вопросы, как комбинаторные алгоритмы, перебор, алгоритмы на графах, алгоритмы вычислительной геометрии. Приводятся избранные олимпиадные задачи по программированию с указаниями к решению. Практические рекомендации по тестированию программ являются необходимым дополнением курса.

Для школьников, студентов и специалистов, серьезно изучающих программирование, а также для преподавателей учебных заведений.

УДК 519.85(023)
ББК 22.18

По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон: (499) 157-5272
e-mail: binom@Lbz.ru, http://www.Lbz.ru

ISBN 978-5-9963-2311-1

Содержание

Единственной женщине посвящаю.

Предисловие

Курс «Программирование в алгоритмах» является естественным продолжением курса «Основы программирования». Его содержание достаточно традиционно: структурное программирование, технологии «сверху — вниз» и «снизу — вверх». Освоению обязательной программы курса автор придает огромное значение. Она обязана стать естественной схемой решения задач, если хотите — культурой мышления или познания.

Только после этого, по нашему глубокому убеждению, разумно переходить на объектно-ориентированное программирование, работу в визуальных средах, на машинно-ориентированное программирование и т. д. Практика подтвердила жизненность данной схемы изучения информатики. Ученики физико-математического лицея г. Кирова многие годы представляют регион на различных соревнованиях по информатике, включая Международные олимпиады. Возвращение их без дипломов или медалей — редкое исключение. Переходя в разряд студентов, выпускники лицея без особых хлопот изучают дисциплины по информатике в любом высшем учебном заведении России, а также успешно выступают в командных чемпионатах мира среди студентов по программированию.

Для кого предназначен учебник? Во-первых, для учителей и учащихся школ с углубленным изучением информатики. Во-вторых, для студентов высших учебных заведений, изучающих программирование и стремящихся достичь профессионального уровня. Особенно он будет полезен тем, кто готовится принять участие в олимпиадах по программированию, включая широко известный чемпионат мира по программированию, проводимый под эгидой международной организации ACM (Association for Computing Machinery).

О благодарностях, ибо не так часто вслух предоставляется возможность сказать коллегам о том, что ты их безмерно уважаешь. Во-первых, это касается моих учеников и учителей одновременно. Без сотрудничества с ними вряд ли автор смог написать что-то подобное. Первая попытка написания книг такого рода была предпринята в 1993 году с Антоном Валерьевичем Лапуновым. То было совместное вхождение в данную проблематику. Для Антона более легкое, для автора достаточно

трудное. Затем последовало сотрудничество с Виталием Игоревичем Беровым. Наверное, благодаря ему автор нашел ту схему изложения алгоритмов на графах, которую он затем применял в работе даже с восьмиклассниками. Сотрудничество с Виктором Александровичем Матюхиным в пору его ученичества было не таким явным, но после того, как он стал студентом МГУ и в настоящее время, оно, на мой взгляд, очень плодотворно. Влияние Виктора на развитие олимпиадной информатики в г. Кирове просто огромно. С братьями Пестовыми, Андреем и Олегом, написаны книги. Более трудолюбивых и отзывчивых учеников автору не приходилось встречать. Сотрудничество с такими ребятами не было бы возможным без высочайшего профессионализма Владислава Владимировича Юферева и Галины Константиновны Корякиной, директора и завуча физико-математического лицея г. Кирова. Особые слова признательности хотелось бы выразить Владимиру Михайловичу Кирюхину, руководителю сборной команды школьников России по информатике. В 1994 году он привлек автора к работе в жюри российской олимпиады школьников. За эти годы автор воочию увидел весь тот тяжелейший труд, который стоит за победами российских школьников на Международных олимпиадах. Иосиф Владимирович Романовский сделал ряд ценных замечаний по газетной версии главы 5, за что автор благодарит его и желает дальнейших творческих успехов в деле обучения петербургских студентов. Многие годы главный редактор газеты «Информатика» Сергей Львович Островский поддерживал автора в его работе, и признательность за это остается неизменной.

Об ошибках. Учебники такого плана не могут не содержать ошибок. Для многих алгоритмов можно, естественно, найти другие схемы реализации. Автор с признательностью примет все замечания. Присылайте их, пожалуйста, по адресу okulov@vspu.kirov.ru

1. Арифметика многоразрядных целых чисел

Известно, что арифметические действия, выполняемые компьютером в ограниченном числе разрядов, не всегда позволяют получить точный результат. Более того, мы ограничены размером (величиной) чисел, с которыми можем работать. А если нам необходимо выполнить арифметические действия над очень большими числами, например
30!= 265252859812191058636308480000000?
В таких случаях мы сами должны позаботиться о представлении чисел и о точном выполнении арифметических операций над ними.

1.1. Основные арифметические операции

Числа, для представления которых в стандартных компьютерных типах данных не хватает количества двоичных разрядов, называются иногда «длинными». В этом случае программисту приходится самостоятельно создавать подпрограммы выполнения арифметических операций. Рассмотрим один из возможных способов их реализации.

Представим в виде:
$30!=2\times(10^4)^8+6525\times(10^4)^7+2859\times(10^4)^6+8121\times(10^4)^5+9105(10^4)^4+$
$+8636\times(10^4)^3+3084\times(10^4)^2+8000\times(10^4)^1+0000\times(10^4)^0$.
Это представление наталкивает на мысль о массиве.

Номер элемента в массиве *A*	0	1	2	3	4	5	6	7	8	9
Значение	9	0	8000	3084	8636	9105	8121	2859	6525	2

Возникают вопросы. Что за 9 в *A*[0], почему число хранится «задом наперед»? Ответы очевидны, но подождем с ними. Будет ясно из текста.

Примечание

Мы работаем с положительными числами!

Первая задача. Ввести число из файла.
Но прежде описание данных.

```
Const MaxDig=1000;{*Максимальное количество цифр –
четырехзначных.*}
Osn=10000;{*Основание нашей системы счисления, в
элементах массива храним четырехзначные числа.*}
Type TLong=Array[0..MaxDig] Of Integer;{*Вычислите
максимальное количество десятичных цифр в нашем числе.*}
```

Прежде чем рассмотреть процедуру ввода, приведем пример. Пусть в файле записано число 23851674 и основанием (*Osn*) является 1000 (храним по три цифры в элементе массива *A*). Изменение значений элементов массива *A* в процессе ввода отражено в таблице 1.1. В основу положен посимвольный ввод, для этого используется переменная *ch*.

Таблица 1.1

A[0]	*A*[1]	*A*[2]	*A*[3]	*ch*	Примечание
3	674	851	23	-	Конечное состояние
0	0	0	0	2	Начальное состояние
1	2	0	0	3	1-й шаг
1	23	0	0	8	2-й шаг
1	238	0	0	5	3-й шаг
2	385	2	0	1	4-й шаг
2	851	23	0	6	5-й шаг
2	516	238	0	7	6-й шаг
3	167	385	2	4	7-й шаг
3	674	851	23		

Итак, в *A*[0] храним количество задействованных (ненулевых) элементов массива *A* — это уже очевидно. И при обработке каждой очередной цифры входного числа старшая цифра элемента массива с номером *i* становится младшей цифрой числа в элементе *i*+1, а вводимая цифра будет младшей цифрой числа из *A*[1].

Примечание (методическое)

Можно ограничиться этим объяснением и разработку процедуры вынести на самостоятельное задание. Можно продолжить объяснение. Например, выписать фрагмент текста процедуры переноса старшей цифры из *A*[*i*] в младшую цифру *A*[*i*+1], т. е. сдвиг уже введенной части на одну позицию вправо:

```
For i:=A[0] DownTo 1 Do
  Begin
    A[i+1]:=A[i+1]+(LongInt(A[i])*10) Div Osn;
    A[i]:=(LongInt(A[i])*10) Mod Osn;
  End;
```

Пусть мы вводим число 23851674 и первые 6 цифр уже разместили «задом наперед» в массиве *A*. В символьную переменную *ch* считали очередную цифру многоразрядного числа — это «7». По нашему алгоритму она должна быть размещена как младшая цифра в *A*[1]. Выписанный фрагмент программы освобождает место для этой цифры. В таблице 1.2 отражены результаты работы этого фрагмента.

Таблица 1.2

i	*A*[1]	*A*[2]	*A*[3]	*ch*
2	516	238	0	7
2	516	380	2	
1	160	385	2	

После этого остается только добавить текущую цифру к *A*[1] и изменить значение *A*[0].

В конечном итоге процедура должна иметь следующий вид:

```
Procedure ReadLong(Var A:TLong);
 Var ch:Char;i:Integer;
 Begin
  FillChar(A,SizeOf(A),0);
  Repeat
   Read(ch);
  Until ch In ['0'..'9']
                {*Пропуск не цифр в начале файла.*}
  While ch In ['0'..'9'] Do
    Begin
      For i:=A[0] DownTo 1 Do
        Begin{*"Протаскивание"
              старшей цифры в числе из A[i] в младшую
              цифру числа из A[i+1].*}
          A[i+1]:=A[i+1]+(LongInt(A[i])*10) Div Osn;
          A[i]:=(LongInt(A[i])*10) Mod Osn;
        End;
```

```
      A[1]:=A[1]+Ord(ch)-Ord('0');
      {*Добавляем младшую цифру к числу из A[1].*}
     If A[A[0]+1]>0 Then Inc(A[0]);
       {*Изменяем длину, число задействованных
        элементов массива A.*}
     Read(ch);
   End;
 End;
```

Вторая задача. Вывод многоразрядного числа в файл или на экран.

Казалось бы, нет проблем — выводи число за числом. Однако в силу выбранного нами представления числа необходимо всегда помнить, что в каждом элементе массива хранится не последовательность цифр числа, а значение числа, записанного этими цифрами. Пусть в элементах массива хранятся четырехзначные числа. И есть число, например, 128400583274. При выводе нам необходимо вывести не 58, а 0058, иначе будет потеря цифр. Итак, нули также необходимо выводить. Процедура вывода имеет вид:

```
Procedure WriteLong(Const A:TLong);
 Var ls,s:String;
     i:Integer;
 Begin
  Str(Osn Div 10,ls);
  Write(A[A[0]]);{*Выводим старшие цифры числа.*}
  For i:=A[0]-1 DownTo 1 Do
    Begin
      Str(A[i],s);
      While Length(s)<Length(ls) Do s:='0'+s;
        {*Дополняем незначащими нулями.*}
      Write(s);
    End;
  WriteLn;
 End;
```

Третья задача. Сложение двух положительных чисел.

Предварительная работа по описанию способа хранения, вводу и выводу многоразрядных чисел выполнена. У нас есть все необходимые «кирпичики», например, для написания программы сложения двух положительных чисел. Исходные числа и результат храним в файлах. Назовем процедуру сложения *SumLongTwo*. Тогда программа ввода двух чисел и вывода результата их сложения будет иметь следующий вид:

```
Var A,B,C:TLong;
Begin
 Assign(Input,'Input.txt'); Reset(Input);
 ReadLong(A);ReadLong(B);
 Close(Input);
 SumLongTwo(A,B,C);
 Assign(Output,'Output.txt'); Rewrite(Output);
 WriteLong(C);
 Close(Output);
End.
```

Трудно ли объяснить процедуру сложения? Используя простой пример — нет.

Пусть A=870613029451, B=3475912100517461.

Результат — C=3476782713546912. Алгоритм имитирует привычное сложение столбиком, начиная с младших разрядов. И именно для простоты реализации арифметических операций над многоразрядными числами используется машинное представление «задом наперед». Ниже приведен текст процедуры сложения двух чисел.

```
Procedure SumLongTwo(Const A,B:TLong;Var C:TLong);
 Var i,k:Integer;
 Begin
  FillChar(C,SizeOf(C),0);
  If A[0]>B[0] Then k:=A[0] Else k:=B[0];
  For i:=1 To k Do
    Begin
     C[i+1]:=(C[i]+A[i]+B[i])Div Osn;
     C[i]:=(C[i]+A[i]+B[i]) Mod Osn;
     {*Есть ли в этих операторах ошибка?*}
    End;
  If C[k+1]=0 Then C[0]:=k Else C[0]:=k+1;
 End;
```

Таблица 1.3

i	*A*[i]	*B*[i]	*C*[1]	*C*[2]	*C*[3]	*C*[4]
1	9451	7461	6912	1	0	0
2	1302	51	6912	1354	0	0
3	8706	9121	6912	1354	7827	1
4	0	3475	6912	1354	7827	3476

Четвертая задача. Реализация операций сравнения чисел: $A=B$, $A<B$, $A>B$, $A\le B$, $A\ge B$.

Функция $A=B$ имеет вид.

```
Function Eq(Const A,B:TLong):Boolean;
Var i:Integer;
Begin
  Eq:=False;
  If A[0]=B[0] Then
    Begin
      i:=1;
      While (i<=A[0]) And (A[i]=B[i]) Do Inc(i);
      Eq:=(i=A[0]+1);
    End;
End;
```

Реализация функции $A>B$ также прозрачна.

```
Function More(A,B:TLong):Boolean;
 Var i:Integer;
 Begin
  If A[0]<B[0]
    Then More:=False
    Else
      If A[0]>B[0]
        Then More:=True
        Else
          Begin
            i:=A[0];
            While (i>0) And (A[i]=B[i]) Do Dec(i);
            If i=0
              Then More:=False
              Else
                If A[i]>B[i]
                  Then More:=True
                  Else More:=False;
          End;
 End;
```

Остальные функции реализуются через функции *Eq* и *More*.

```
Function Less(A,B:TLong):Boolean;{A<B}
 Begin
  Less:=Not(More(A,B) Or Eq(A,B));
 End;
```

```
Function More_Eq(A,B:TLong):Boolean;
 Begin
  More_Eq:=More(A,B) Or Eq(A,B);
 End;
```

И наконец, последняя функция $A \le B$.

```
Function Less_Eq(A,B:TLong):Boolean;
 Begin
  Less_Eq:=Not(More(A,B));
 End;
```

Для самостоятельного решения может быть предложена следующая более сложная задача. Требуется разработать функцию, которая выдает 0, если A больше B, 1, если A меньше B и 2 при равенстве чисел. Но сравнение должно быть выполнено с учетом сдвига. О чем идет речь? Поясним на примере.

Пусть A равно 56784, а B — 634. При сдвиге на 2 позиции влево числа B функция должна сказать, что B больше A, без сдвига — что A больше B. Или A равно 567000, а B — 567 и сдвиг равен 3, то функция должна «сказать», что числа равны.

Решение может иметь следующий вид.

```
Function More(Const A, B: TLong; sdvig: Integer):
  Byte;
 Var i: Integer;
  Begin
   If A[0]>(B[0]+sdvig)
    Then More:=0
    Else
     If A[0]<(B[0]+sdvig)
      Then More:=1
      Else
       Begin
        i:=A[0];
        While (i>sdvig) And (A[i]=B[i-sdvig]) Do Dec(i);
        If i=sdvig
         Then
          Begin
           More:=0;{*Совпадение чисел с учетом
                    сдвига.*}
           For i:=1 To sdvig Do
            If A[i]>0 Then Exit;
              More:=2;{*Числа равны, "хвост" числа A
                       равен нулю.*}
```

```
        End
       Else More:=Byte(A[i]<B[i-sdvig]);
     End;
  End;
```

Пятая задача. Умножение многоразрядного числа на короткое.

Под коротким числом понимается целое число, не превосходящее основание системы счисления.

Процедура очень похожа на процедуру сложения двух чисел.

```
Procedure Mul(Const A: TLong;Const K: LongInt;
              Var C: TLong);
 Var i: Integer;
{*Результат - значение переменной C.*}
  Begin
   FillChar(C, SizeOf(C), 0);
   If K=0
    Then Inc(C[0]){*Умножение на ноль.*}
    Else
      Begin
       For i:=1 To A[0] Do
        Begin
         C[i+1]:=(LongInt(A[i])*K+C[i]) Div Osn;
         C[i]:=(LongInt(A[i])*K +C[i]) Mod Osn;
        End;
        If C[A[0]+1]>0
          Then C[0]:=A[0]+1
          Else C[0]:=A[0];
           {*Определяем длину результата.*}
      End;
  End;
```

В качестве *самостоятельной работы* можно предложить разработку процедуры умножения двух чисел. Возможный вариант процедуры приведен ниже.

```
Procedure MulLong(Const A, B: TLong; Var C: TLong);
{*Умножение "длинного" на "длинное".*}
 Var i, j: Word;
       dv: LongInt;
 Begin
  FillChar(C, SizeOf(C), 0);
  For i:=1 To A[0] Do
```

```
 For j:=1 To B[0] Do
  Begin
   dv:=LongInt(A[i])*B[j] + C[i+j-1];
   Inc(C[i+j], dv Div Osn);
   C[i+j-1]:=dv Mod Osn;
  End;
 C[0]:=A[0]+B[0];
 While (C[0]>1) And (C[C[0]]=0) Do Dec(C[0]);
End;
```

Шестая задача. Вычитание двух многоразрядных чисел с учетом сдвига.

Если суть сдвига пока не понятна, то оставьте ее в покое, на самом деле вычитание с учетом сдвига потребуется при реализации операции деления. Выясните алгоритм работы процедуры при нулевом сдвиге.

Введем ограничение: число, из которого вычитают, больше числа, которое вычитается.

Работать с длинными отрицательными числами мы не умеем. Процедура была бы похожа на процедуры сложения и умножения, если бы не одно «но» — не перенос единицы в старший разряд, а «заимствование единицы из старшего разряда». Например, в обычной системе счисления мы вычитаем 9 из 11 — идет заимствование 1 из разряда десятков, а если из 10000 той же 9 — процесс заимствования несколько сложнее.

```
Procedure Sub(Var A: TLong;Const B: TLong;
              sp: Integer);
 Var i,j: Integer;{*Из A вычитаем B с учетом
           сдвига sp, результат вычитания в A.*}
  Begin
   For i:=1 To B[0] Do
      Begin
         Dec(A[i+sp], B[i]);
         {*Реализация сложного заимствования*} {*)
         j:=i;                                 {*)
         While (A[j+sp]<0) And (j<=A[0]) Do
          Begin                                {*)
           Inc(A[j+sp], Osn);Dec(A[j+sp+1]);Inc(j); {*}
          End;                                 {*}
      End;
   i:=A[0];
   While (i>1) And (A[i]=0) Do Dec(i);
```

```
    A[0]:=i; (*Корректировка длины результата
              операции*)
End;
```

Если строки, отмеченные {*}, заменить на нижеприведенные операторы:

```
{*Реализация простого заимствования*}
If A[i+sp]<0 Then
   Begin
     Inc(A[i+sp], Osn);
     Dec(A[i+sp+1]);
   End;
```

то, по понятным причинам, алгоритм не будет работать при всех исходных данных. Можно сознательно сделать ошибку и предложить найти ее — принцип «обучение через ошибку».

Рекомендуется выполнить трассировку работы данной процедуры, например, для следующих исходных данных.

A=10000001000000000000,
B=2000073859998.

Седьмая задача. Деление двух многоразрядных чисел, т. е. нахождение целой части частного и остатка.

Написать исходный (без уточнений) алгоритм не составляет труда:

```
Procedure LongDivLong(Const A, B: Tlong;
                      Var Res, Ost: TLong);
  Begin
    FillChar(Res, SizeOf(Res), 0); Res[0]:=1;
    FillChar(Ost, SizeOf(Ost), 0); Ost[0]:=1;
    Case More(A, B, 0) Of
     0: MakeDel(A, B, Res, Ost); {*A больше B, пока
                не знаем, как выполнять операцию, -
               "выносим" в процедуру.*}
     1: Ost:=A; {*A меньше B.*}
     2: Res[1]:=1; {*A равно B.*}
    End;
  End;
```

А дальше? Дальше начинаются проблемы. Делить столбиком нас научили в школе. Например:

```
 1000143123567 |73859998
-  73859998    |  13541(Целая часть частного)
  261543143
- 221579994
  399631495
- 369299990
  303315056
- 295439992
   78750647
-  73859998
    4890649 (Остаток)
```

Что мы делали? На каждом этапе в уме подбирали цифру (1, 3, 5 и т. д.), такую, что произведение этой цифры на делитель дает число меньшее, но наиболее близкое к числу. Какому? Это трудно сказать словами, но из примера ясно. Зачем нам это делать в уме? Пусть делает компьютер. Однако упростим пример, оставим его для тестирования окончательной логики, тем более что и числа длинные. Пусть число A будет меньше $B{\times}10$, тогда в результате (целая часть деления) будет одна цифра. Например, A равно 564, а B — 63 и простая десятичная система счисления. Попробуем подобрать цифру результата, но не методом прямого перебора, а методом деления отрезков пополам. Пусть *Down* — нижняя граница интервала изменения подбираемой цифры, *Up* — верхняя граница интервала, *Ost* равен делимому.

Таблица 1.4

Down	*Up*	*C=B*((Down+Up) Div 2)*	*Ost*=564
0	10	315	*C<Ost*
5	10	441	*C<Ost*
7	10	504	*C<Ost*
8	10	567	*C>Ost*
8	9	504	*C<Ost*

Итак, результат — целая часть частного равна (*Down*+*Up*)*div 2*, остаток от деления — разность между значениями *Ost* и *C*. Нижнюю границу *Down* изменяем, если результат (*C*) меньше остатка, верхнюю *Up*, если больше. В таблице 1.4 приведены результаты вычислений.

Усложним пример. Пусть A равно 27856, а B — 354. Основанием системы счисления является не 10, а 10000. В таблице 1.5 приведено поэтапное решение этой задачи.

Таблица 1.5

Down	*Up*	*C*	*Ost*=27856
0	10000	1770000	*C>Ost*
0	5000	885000	*C>Ost*
0	2500	442500	*C>Ost*
0	1250	221250	*C>Ost*
0	625	110448	*C>Ost*
0	312	55224	*C>Ost*
0	156	27612	*C<Ost*
78	156	41418	*C>Ost*
78	117	34338	*C>Ost*
78	97	30798	*C>Ost*
78	87	29028	*C>Ost*
78	82	28320	*C>Ost*
78	80	27966	*C>Ost*
78	79	27612	*C<Ost*

Целая часть частного равна 78, остаток от деления — 27856 минус 27612, т. е. 244.

Пора приводить процедуру. Используемые «кирпичики»: функция сравнения чисел (*More*) с учетом сдвига и умножения многоразрядного числа на короткое (*Mul*) описаны выше.

```
Function FindBin(Var Ost: TLong;Const B: Tlong;
                 sp: Integer): LongInt;
  Var Down, Up: Word;
      C: TLong;
   Begin
    Down:=0;Up:=Osn;{*Основание системы счисления.*}
    While Up-1>Down Do
Begin {*У преподавателя есть возможность сделать
сознательную ошибку. Изменить условие цикла на
Up>Down. Результат - зацикливание программы.*}
      Mul(B, (Up+Down) Div 2, C);
       Case More(Ost, C, sp) Of
         0: Down:=(Down+Up) Div 2;
         1: Up:=(Up+Down) Div 2;
         2: Begin Up:=(Up+Down) Div 2; Down:=Up; End;
```

```
    End;
  End;
  Mul(B, (Up+Down) Div 2, C);
  If More(Ost,C,0)=0
    Then Sub(Ost, C, sp)
           {*Находим остаток от деления.*}
    Else
      Begin Sub(C,Ost,sp); Ost:=C; End;
  FindBin:=(Up+Down) Div 2;{*Целая часть частного.*}
End;
```

Осталось разобраться со сдвигом — значением переменной *sp* в нашем изложении.

Вернемся к обычной системе счисления и попытаемся разделить числа, например, 635 на 15. Что мы делаем? Вначале делим 63 на 15 и формируем, подбираем в уме, первую цифру результата. Подбирать с помощью компьютера мы научились. Подобрали — это цифра 4, и это старшая цифра результата. Изменим остаток. Если вначале он был 635, то сейчас стал 35. Вычитать с учетом сдвига мы умеем. Опять подбираем цифру. Вторую цифру результата. Это цифра 2 и остаток 5. Итак, результат (целая часть) 42, остаток от деления 5. А что изменится, если основанием будет не 10, а 10000? Алгоритм совпадает, только в уме считать несколько труднее, но ведь у нас же есть молоток под названием компьютер — пусть он вбивает гвозди.

```
Procedure MakeDel(Const A, B: Tlong;
                  Var Res, Ost:TLong);
 Var sp: Integer;
  Begin
    Ost:=A;{*Первоначальное значение остатка.*}
    sp:=A[0]-B[0];
    If More(A, B, sp)=1 Then Dec(sp);
    {* B*Osn>A, в результате одна цифра.*}
    Res[0]:=sp+1;
    While sp>=0 Do
      Begin
        {*Находим очередную цифру результата.*}
        Res[sp+1]:=FindBin(Ost, B, sp);
        Dec(sp);
      End;
  End;
```

Примечание

В результате работы по данной теме каждый учащийся обязан иметь собственный модуль для работы с многоразрядными числами, который должен содержать следующий (примерный) перечень функций и процедур (описательная часть). Курсивом выделены процедуры и функции, не рассмотренные в данной главе.

```
Function Eq(Const A, B: TLong): Boolean;
{*Сравнение чисел A=B*}
Function Less(Const A, B: TLong): Boolean;
{*Сравнение чисел A<B*}
Function More(Const A, B: TLong): Boolean;
{*Сравнение чисел A>B*}
Function Less_Eq(Const A, B: TLong):Boolean;
{*Сравнение чисел A≤B*}
Function More_Eq(Const A, B: TLong): Boolean;
{*Сравнение чисел A≥B*}

Function LongTurnShort(Const A: TLong;
Var K: LongInt): Boolean;
{*Преобразование многоразрядного числа в число
типа LongInt.*}

Function LongModShort(Const A: TLong;
Const K: Word): Word;
{*Остаток от деления многоразрядного числа на
короткое типа Word.*}

Function HowDigits(Const A: TLong): LongInt;
{*Подсчет числа десятичных цифр в многоразрядном
числе.*}

Procedure Mul(Const A: TLong; K: Word;
Var B: TLong);
{*Умножение многоразрядного числа на короткое типа
Word.*}

Procedure MulLong(Const A, B: TLong; Var C: TLong);
{*Умножение двух многоразрядных чисел*}

Procedure SumLongTwo(Const A, B: TLong;
Var C: TLong);
{*Сложение двух многоразрядных чисел*}

Procedure Sub(Var A: TLong;Const B: Tlong;
Const sp:Integer);
{*Вычитание многоразрядных чисел*}
```

```
Procedure ShortTurnLong(K: LongInt; Var A: TLong);
{*Преобразование числа типа LongInt в многоразрядное
число.*}

Procedure LongDivShort(Const A: TLong;
Const K: Word; Var B: TLong);
{*Целочисленное деление многоразрядного числа на
короткое типа Word.*)

Procedure LongDivLong(Const A, B: TLong;
Var C, ost: TLong);
{*Деление двух многоразрядных чисел*}

Procedure SdvigLong(Var A: TLong; Const p: Word);
{*Сдвиг влево на p длинных цифр.*}
```

1.2. Задачи

1. Для представления многоразрядных чисел использовался одномерный массив. Каждая цифра числа (или группа цифр) хранилась в отдельном элементе массива. Рассмотрим представление многоразрядных чисел в виде двусвязного списка. Каждая цифра числа — элемент списка. На рисунке 1.1 представлено число 25974.

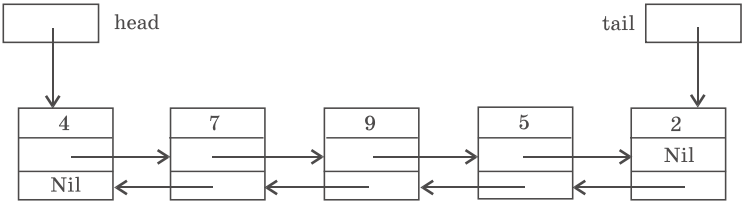


Рис. 1.1

Число хранится «задом наперед». Такое представление удобно при выполнении арифметических операций, а для вывода у нас есть адрес предшествующего элемента. Элемент списка можно описать следующим образом:

```
Const Basis=10;{*Основание системы счисления.*}
 Typept=^elem;
   elem=Record
          data:Integer;
          next,pred:pt
        End;
 Var head, tail :pt;
```

Разработать программы выполнения основных арифметических операций при таком способе хранения многоразрядных чисел.

2. Написать программу вычисления $N!$, при $N \geq 100$.

3. Написать программу вычисления 3^N, при $N \geq 100$.

4. Написать программу извлечения квадратного корня из многоразрядного числа.

Указание. Пусть требуется найти целую часть результата. Возможна следующая схема решения задачи [9].

Шаг 1. Многоразрядное число разбивается на несколько пар (граней) чисел. Если число содержит четное количество разрядов, тогда первая грань содержит две цифры, если нечетное, то одну. Так, число 236488 разбивается на три грани, каждая из которых содержит две цифры: 23 64 88, а для числа 15385 первая грань состоит из одной цифры 1.

Число граней определяет количество разрядов ответа. В качестве примера возьмем число 574564 и разобьем его на грани: 57 45 64. Таким образом, ответом является трехразрядное число (обозначим через y).

Шаг 2. Рассматривается первая грань 57 и подбирается число x такое, чтобы x^2 было как можно ближе к 57, но не превосходило его. Таким образом, $x=7$, так как $7\times7=49<57$, а $8\times8=64>57$. Первая цифра ответа y равна 7.

Шаг 3. Находится разность между первой гранью 57 и $x^2=49$: $57-49=8$. Эта разность приписывается к следующей грани, получается число 845.

Шаг 4. Подбирается следующая цифра. Находится новое значение x, такое, чтобы $(2\times y\times10+x)\times x$ не превосходило 845 и было наиболее близким к 845. Это число 5, так как $(14\times10+5)\times5=725<845$, а $(14\times10+6)\times6=876>845$. Итак, вторая цифра ответа 5 и $y=75$.

Шаг 5. Повторятся шаг 3. Находим разность: $845-725=120$, ее «подклеим» к следующей грани — 12064.

Шаг 6. Выполняются действия шага 4. $(2\times y\times10+x)\times x<=12064$. Значение x равно 8, так как $(150\times10+8)\times8=12064$. Итак, $y=758$.

Пояснений требует шаг 4, а именно, не понятно, почему используется такая формула для подбора очередной цифры ответа.

Пусть $10\times y+x=\sqrt{z}$. Возведем в квадрат число $10\times y+x$. Получим $100\times y^2+10\times y\times x+10\times y\times x+x^2=z$.

Значение y подобрано на шаге 2, оно дает вклад $100\times y^2$ в число z. Подбор цифры x обязан дать максимальный вклад в число $z-100\times y^2$.

Примечание

Если корень не извлекается в результате повторения шагов 3, 4, то после последней цифры заданного числа ставят запятую и образуют дальнейшие грани, каждая из которых имеет вид 00. В этом случае процесс извлечения корня прекращается при достижении требуемой точности.

5. Определить, встречаются ли в записи некоторого многоразрядного числа две подряд идущие цифры, например 9?

6. Определить наибольший общий делитель двух многоразрядных чисел.

7. Определить наименьшее общее кратное двух многоразрядных чисел.

8. Определить, являются ли два многоразрядных числа взаимно простыми.

9. Определить количество делителей многоразрядного числа.

10. Написать программу проверки того, что число $2^{19936}\times(2^{19937}-1)$ является совершенным, т. е. равным сумме своих делителей, кроме самого себя.

11. Написать программу перевода многоразрядного числа в системы счисления с основаниями 2, 8, 16.

12. Написать программу разложения многоразрядного числа на простые множители.

13. Известно, что целочисленное деление (умножение) на 2 равносильно сдвигу числа на один разряд вправо (влево). Два обычных числа a и b можно сложить с помощью следующего алгоритма:

- если a и b четные числа, то $a+b=\left(\frac{a}{2}+\frac{b}{2}\right)\times 2$;
- если a и b нечетные числа, то $a+b=\left(\frac{a-1}{2}+\frac{b-1}{2}+1\right)\times 2$;
- если a — четное, а b — нечетное, то $a+b=\left(\frac{a}{2}+\frac{b-1}{2}\right)\times 2+1$;
- если a — нечетное, а b — четное, то $a+b=\left(\frac{a-1}{2}+\frac{b}{2}\right)\times 2+1$.

Заметим, что при нечетном значении a для целочисленного деления $\frac{a-1}{2} \equiv \frac{a}{2}$.

Приведенный алгоритм выполнения операции сложения реализуется с помощью следующей рекурсивной функции:

```
Function Sum(x,y:Integer):Integer;
 Begin
  If x=0
   Then Sum:=y
   Else
    If y=0
     Then Sum:=x
     Else
       If (x Mod 2 =0) And (y Mod 2 =0)
         Then Sum:=Sum(x Div 2, y Div 2)*2
         Else
           If (x Mod 2 =1) And (y Mod 2 =1)
             Then Sum:=Sum(x Div 2,y Div 2 +1)*2
             Else Sum:=Sum(x Div 2,y Div 2)*2+1;
 End;
```

Написать аналогичную процедуру сложения (не обязательно рекурсивную) двух многоразрядных целых чисел a и b. Деление и умножение на 2 заменить соответствующими операциями сдвига на один разряд.

14. Два числа a и b можно перемножить, используя только сдвиги на один разряд и операцию сложения.

Например, b=125, a=37.

Выпишем следующий столбик чисел:

125×37
250×18+125
500×9+125
1000×4+500+125
2000×2+500+125
4000×1+500+125.

И результат: 125×37=125+500+4000=4625 (в каких случаях изменяемое число b входит в сумму?). Он основан на следующих тождествах:

- если a четное число, то $a \times b = \left(\frac{a}{2} \times b\right) \times 2$;
- если a нечетное число, то $a \times b = \left(\frac{a-1}{2} \times b\right) \times 2 + b$.

Этот алгоритм реализуется с помощью следующей рекурсивной функции:

```
Function RecMul(x,y:Integer):Integer;
(*Естественно предполагается, что результат не
выходит за пределы Integer.*)
 Begin
  If x=0
   Then RecMul:=0
   Else
     If x Mod 2=0
       Then RecMul:=RecMul(x Div 2,y)*2
       Else RecMul:=RecMul(x Div 2,y)*2+y;
End;
```

Написать аналогичную процедуру умножения (не обязательно рекурсивную) двух многоразрядных целых чисел a и b. Деление и умножение на 2 заменить соответствующими операциями сдвига на один разряд.

15. Написать программу сложения двух знаковых многоразрядных чисел [2].

Указание. Необходимо заменить t-разрядное отрицательное число $numb$ на его дополнение до числа 10^t, то есть на число $10^t - |numb|$.

Алгоритм получения дополнительного кода отрицательного числа известен [2, 23]. Находят обратный код десятичного числа путем замены каждой цифры i на $9-i$. Прибавление единицы к обратному коду дает дополнительный код. После замены отрицательных слагаемых на их дополнительный код работает алгоритм сложения многоразрядных чисел, рассмотренный в 1.1.

2. Комбинаторные алгоритмы

Одна из главных целей изучения темы, помимо традиционных, заключается в том, чтобы учащиеся осознали суть «отношения порядка» на некотором множестве объектов.

На множестве объектов типа *Word, Integer, Char, Boolean* отношение порядка воспринимается как нечто очевидное. Объяснить, что на типе *Real* его нет, гораздо сложнее. Однако осознание того факта, что компьютер что-то может делать с каким-то множеством объектов только в том случае (по крайней мере, перебрать их все), если на этом множестве введено какое-то отношение порядка, требует длительной работы. А выработка автоматических навыков «вычленения» отношения порядка на множестве данных конкретной задачи — работа многих дней.

2.1. Классические задачи комбинаторики

При решении практических задач часто приходится выбирать из некоторого конечного множества объектов подмножество элементов, обладающих теми или иными свойствами, размещать элементы множеств в определенном порядке и т. д. Эти задачи принято называть комбинаторными задачами.

Многие из комбинаторных задач решаются с помощью двух основных правил — суммы и произведения. Если удается разбить множество объектов на классы и каждый объект входит в один и только один класс, то общее число объектов равно сумме числа объектов во всех классах (*правило сложения*). *Правило умножения* чуть сложнее. Даны два произвольных конечных множества объектов A и B. Количество объектов в A равно N, в B — M. Составляются упорядоченные пары ab, где $a \in A$ (a принадлежит A) и $b \in B$. Количество таких пар (объектов) равно $N \times M$. Правило обобщается и на большее количество множеств объектов.

Перестановки (P_N — число перестановок).

Классической задачей комбинаторики является задача о числе перестановок — сколькими способами можно переставить N различных предметов, расположенных на N различных местах.

Объяснение примеров

1. Сколькими способами можно переставить три монеты 1, 2, 5 рублей, расположенных соответственно на трех местах с номерами 1, 2, 3? Ответ: 6.
2. Сколькими способами можно пересадить четырех гостей А, Б, В, Г, сидящих соответственно на четырех местах 1, 2, 3, 4? Ответ: 24.
3. Сколькими способами можно переставить буквы в слове «*эскиз*»? Ответ: 120.
4. Сколькими способами можно расположить на шахматной доске 8 ладей так, чтобы они не могли бить друг друга? Ответ: 40320.

N различных предметов, расположенных на N различных местах, можно переставить

$N!$ (N-факториал) $= 1\times2\times3\times...\times N$ способами ($P_N=N!$).

Отвлекаясь от природы объектов, можно считать их для простоты натуральными числами от 1 до N. При этой трактовке понятие перестановки можно дать следующим образом. Перестановкой из N элементов называется упорядоченный набор из N различных чисел от 1 до N. Места также нумеруются от 1 до N. Тогда перестановку можно записать таблицей с двумя строками, из которых каждая содержит все числа от 1 до N.

Например, $F=\begin{pmatrix}1&2&3&4&5&6\\3&5&1&6&4&2\end{pmatrix}$.

Перестановку

$E=\begin{pmatrix}1&2&3&4&\dots&N\\1&2&3&4&\dots&N\end{pmatrix}$ называют тождественной.

По перестановке F однозначно определяется перестановка F^{-1}, такая, что $FF^{-1}=E$.

Так, для нашего примера $F^{-1}=\begin{pmatrix}1&2&3&4&5&6\\3&6&1&5&2&4\end{pmatrix}$.

Под суперпозицией перестановок F и G понимают перестановку FG, определяемую следующим образом: $FG(i)=F(G(i))$.

Так, если F совпадает с приведенной выше перестановкой, а

$G=\begin{pmatrix}1&2&3&4&5&6\\6&3&5&1&3&2\end{pmatrix}$, то $FG=\begin{pmatrix}1&2&3&4&5&6\\2&1&4&3&1&5\end{pmatrix}$.

Каждую перестановку можно представить графически в виде ориентированного графа с множеством вершин от 1 до N.

Для перестановки $F = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 1 & 6 & 4 & 2 \end{pmatrix}$ соответствующий ориентированный граф изображен на рисунке 2.1.

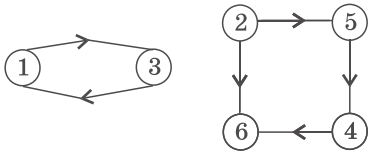


Рис. 2.1

Графическая иллюстрация облегчает введение понятия цикла перестановки. Видим, что перестановка F представима двумя циклами: F=[1, 3], [2, 5, 4, 6].

Пару f_i и f_j, (где $i<j$) называют *инверсией* перестановки F, если $f_i>f_j$. Число инверсий (K) определяет знак перестановки $(-1)^K$. Для нашего примера число инверсий равно 6, а знак перестановки положительный. Произвольную перестановку, являющуюся циклом длины 2, обычно называют *транспозицией*.

Размещения (A_N^M).

Задача формулируется следующим образом: сколькими способами можно выбрать и разместить по M различным местам M из N различных предметов?

Объяснение примеров

1. Сколькими способами можно выбрать и разместить на двух местах 1, 2 две из трех монет 1, 2, 5 рублей? Ответ: 6.
2. Сколькими способами можно выбрать и разместить на двух местах 1, 2 двух из четырех гостей А, Б, В, Г? Ответ: 12.
3. Сколько трехбуквенных словосочетаний можно составить из букв слова *эскиз?* Ответ: 60.
4. В чемпионате по футболу принимают участие 17 команд и разыгрываются золотые, серебряные и бронзовые медали. Сколькими способами они могут быть распределены? Ответ: 4080.
5. Партия состоит из 25 человек. Требуется выбрать председателя партии, его заместителя, секретаря и казначея. Сколькими способами можно это сделать, если каждый член партии может занимать лишь один пост. Ответ: 303600.

Число размещений вычисляется по формуле

$$A_N^M = N \times (N-1) \times (N-2) \times \ldots \times (N-M+1),$$

$$\text{или } A_N^M = \frac{N!}{(N-M)!}$$

Сочетания (выборки) $\left(C_N^M\right)$.

Другой классической задачей комбинаторики является задача о числе выборок: сколькими способами можно выбрать M из N различных предметов?

Объяснение примеров

1. Сколькими способами можно выбрать две из трех монет 1, 2, 5 рублей? Ответ: 3.
2. Сколькими способами можно выбрать двух из четырех гостей А, Б, В, Г? Ответ: 6.
3. Сколькими способами можно выбрать три из пяти букв слова *эскиз*? Ответ: 10.
4. В чемпионате по футболу России принимают участие 17 команд и разыгрываются золотые, серебряные и бронзовые медали. Нас не интересует порядок, в котором располагаются команды-победительницы, ибо все они выходят на Европейские турниры. Сколько различных способов представления нашего государства на Европейских турнирах? Ответ: 680.
5. Партия состоит из 25 человек. Требуется выбрать председателя партии и его заместителя, ибо они представляют партию в президиуме межпартийных форумов. Сколькими способами можно это сделать, если каждый член партии может занимать лишь один пост? Ответ: 300.
6. Сколькими способами можно поставить на шахматной доске 8 ладей (условие о том, что ладьи не могут бить друг друга, снимается)? Ответ: 4328284968.
7. Сколько различных прямоугольников можно вырезать из клеток доски размером $N \times M$ (стороны прямоугольников проходят по границам клеток)?

   Пояснение. Горизонтальные стороны могут занимать любое из $M+1$ положений. Число способов их выбора — C_{M+1}^2, аналогично для вертикальных сторон. Ответ: $C_{M+1}^2 \times C_{N+1}^2$.

Сочетание можно определить и как подмножество из M различных натуральных чисел, принадлежащих множеству чисел из интервала от 1 до N. Число сочетаний вычисляется по формуле

$$C_N^M = \frac{N!}{M! \times (N-M)!}.$$

Полезные формулы:

$C_N^0 = C_N^N = 1$ ($N \geq 1$)

$C_N^M = C_N^{N-M}$ (правило симметрии)

$C_N^M = C_{N-1}^{M-1} + C_{N-1}^M$ ($N>1$, $0<M<N$) (правило Паскаля)

$C_N^M \times C_M^K = C_N^K \times C_{N-K}^{M-K}$

$\sum_{i=0}^{N} C_N^i = 2^N$. Эта формула является следствием известного соотношения под названием бином Ньютона:

$$(x+y)^N = \sum_{k=0}^{N} C_N^k x^k y^{N-k}.$$

При $x=y=1$ бином Ньютона дает приведенную формулу.

Размещения с повторениями (A_N^k).

Даны предметы, относящиеся к N различным типам. Из них составляются всевозможные расстановки длины k. При этом в расстановки могут входить и предметы одного типа. Две расстановки считаются различными, если они отличаются или типом входящих в них предметов, или порядком этих предметов.

Объяснение примеров

1. Найти общее количество шестизначных чисел.
   Ответ: 9×10×10×10×10×10=900000.
2. Найти число буквосочетаний длины 4, составленных из 33 букв русского алфавита, и таких, что любые две соседние буквы этих буквосочетаний различны.
   Ответ: 33×32×32×32=1081344.
3. Ячейка памяти компьютера состоит из 16 бит (k). В каждом бите, как известно, можно записать 1 или 0 (N). Сколько различных комбинаций 1 и 0 может быть записано в ячейке? Ответ: 2^{16}=65536.

Общая формула для подсчета числа размещений с повторениями

$$A_N^k = N^k.$$

Перестановки с повторениями.

Выше рассмотрены перестановки из различных предметов. В том случае, когда некоторые переставляемые предметы одинаковы, часть перестановок совпадает друг с другом. Перестановок становится меньше. Задача формулируется следующим

образом. Имеются предметы k различных типов. Сколько перестановок можно сделать из N_1 предметов первого типа, N_2 — второго, ..., N_k — k-го типа ($N_1+N_2+...+N_k=N$)?

Объяснение примеров

1. Сколько различных буквосочетаний можно получить из букв слова *папа*? Ответ: 6 (папа, паап, ппаа, апап, аапп, аппа).
2. Сколько различных буквосочетаний можно получить из букв слова *капкан*? Ответ: 180.
3. Сколькими способами можно расположить в ряд 2 одинаковых яблока и 3 одинаковые груши? Ответ: 10.
4. Сколько различных сообщений можно закодировать, меняя порядок 6 флажков: 2 красных, 3 синих и 1 зеленый? Ответ: 60.
5. Для выполнения работы необходимо в некотором порядке выполнить дважды каждое из трех действий (a, b, c). Например: *aabbcc*, *cabbac*. Сколько существует различных способов выполнения работы? Ответ: 90.

Общая формула для подсчета перестановок с повторениями имеет вид:

$$P(N_1,\ N_2,\ ...,\ N_k) = \frac{N!}{N_1! \times N_2! \times \ldots \times N_k!},$$

где $N=N_1+N_2+...+N_k$ — общее число элементов.

Эта формула следует из

$$P(N_1,\ N_2,\ ...,\ N_k) = C_N^{N_1} \times C_{N-N_1}^{N_2} \times \ldots \times C_{N-N_1-\ldots N_k}^{N_k}.$$

Сочетания с повторениями.

Имеются предметы N различных типов. Сколько существует расстановок длины k, если не различать порядок предметов в расстановке (различные расстановки должны отличаться хотя бы одним предметом)?

Схема рассуждений. Пусть $N=4$ и $k=7$. Закодируем расстановку с помощью последовательности из нулей и единиц. Запишем столько единиц, сколько предметов первого типа, затем нуль, а затем столько единиц, сколько предметов второго типа, и т. д. (нуль, единицы, соответствующие предметам третьего типа, нуль и единицы для предметов четвертого типа). Суммарное количество единиц в последовательности равно 7, количество нулей — $N-1$ или 3, т. е. длина равна 10. Количество таких последовательностей определяется количеством способов записи 3 нулей на 10 мест, или $C_{10}^3=120$.

Общая формула для подсчета числа сочетаний с повторениями имеет вид

$$\overline{C}_N^k = C_{N+k-1}^{N-1} = C_{N+k-1}^k .$$

Объяснение примеров

1. Четверо ребят собрали в лесу 30 белых грибов. Сколькими способами они могут разделить их между собой? Ответ: $C_{4+30-1}^3 = 5301$.
2. Сколькими способами можно выбрать 3 рыбины из трех одинаковых щук и трех одинаковых окуней? Ответ: 4 (N=2, k=3).
3. Сколькими способами можно выбрать 13 из 52 игральных карт, различая их только по масти? Ответ: 560 (N=4, k=13).
4. Сколько всего существует результатов опыта, заключающегося в подбрасывании 2 одинаковых игральных костей (если кости различные, то 36)? Ответ: 21 (N=6, k=2).
5. Сколько существует целочисленных решений уравнения $x_1+x_2+x_3+x_4=7$? Ответ: 120 (N=4, k=7, пример решения — 1101101101).

Разбиения.

Требуется подсчитать число разбиений конечного множества предметов S, где $|S|=N$ (количество предметов), на k различных подмножеств $S=S_1 \cup S_2 \cup ... \cup S_k$, попарно не пересекающихся, $|S_i|=n_i$, i=1, 2, ..., k и $n_1+n_2+...+n_k=N$.

Схема рассуждений. S_1 можно сформировать $C_N^{N_1}$ способами, S_2 – $C_{N-N_1}^{N_2}$ способами, ... S_k – $C_{N-N_1-...-N_{k-1}}^{N_k}$ способами. Таким образом, число разбиений равно

$$C_N^{N_2} \times C_{N-N_1}^{N_2} \times ... \times C_{N-N_1-...-N_{k-1}}^{N_k},$$

$$\text{или } \frac{N!}{N_1! \times N_2! \times ... \times N_k!}.$$

Объяснение примеров

1. Сколькими способами можно разбить три монеты 1, 2, 5 рублей на три группы по одной монете? Ответ: 6.
2. Сколькими способами можно разбить четырех гостей А, Б, В, Г на две группы? Ответ: 6.

3. Сколькими способами можно разбить пять букв слова *эскиз* на три группы так, чтобы в первой группе была одна буква, а во второй и третьей по 2? Ответ: $\frac{5!}{1! \times 2! \times 2!} = 30$.

4. Сколькими способами можно расселить 8 студентов по трем комнатам: одноместной, трехместной и четырехместной? Ответ: $\frac{8!}{1! \times 3! \times 4!} = 280$.

5. Сколькими способами можно раскрасить в красный, зеленый и синий цвет по 2 из 6 различных предметов? Ответ: $\frac{6!}{2! \times 2! \times 2!} = 90$.

Число Стирлинга второго рода $S(N,k)$ определяется как число разбиений N-элементного множества на k подмножеств. Так, $S(4,2)=7$: {[1, 1, 3], [4]}, {[1, 2, 4], [3]}, {[1, 3, 4], [2]}, {[1, 2], [3, 4]}, {[1, 3], [2, 4]}, {[1, 4], [2, 3]} и {[1], [2, 3, 4]}.

Считается, что $S(0,0)=1$.

Если $k>N$, то $S(N,k)=0$.

При $k=N$ $S(N,N)=1$.

В общем случае справедлива формула

$S(N,k)=S(N-1,k-1)+kS(N-1,k)$ для $0<k<N$.

2.2 Генерация комбинаторных объектов

2.2.1. Перестановки

Первая задача. Вычислить число перестановок для заданного значения N.

Известно, что число перестановок равно факториалу числа N ($N!=1\times2\times3\times...\times N$). В таблице 2.1 приведены значения $N!$ для N от 1 до 30.

Таблица 2.1

N	$N!$
1	1
2	2
3	6
4	24
5	120
6	720

7	5040(Последнее число типа *Integer*)
8	40320
9	362880
10	3628800
11	39916800
12	479001600(Последнее число типа *LongInt*)
13	6227020800
...	
30	265252859812191058636308480000000

Рекурсивное определение: $N!=N\times(N-1)!$

Функция вычисления $N!$ имеет ограниченную область применения.

```
Function Fac(k:LongInt):LongInt;
 Var    r,i: LongInt;
 Begin
   r:=1;
   For i:=1 To k Do r:=r*i;
   Fac:=r;
 End;
```

Функция *Fac* работоспособна при N, меньшем или равном 12. Для бóльших значений N требуется использовать «длинную» арифметику: умножение «длинного» числа на короткое; вывод «длинного» числа (глава 1).

Вторая и третья задачи. Перечислить или сгенерировать все перестановки для заданного значения N, что требует введения отношения порядка на множестве перестановок. Только в этом случае задача имеет смысл.

Предполагаем, что перестановка хранится в массиве P длины N. Лексикографический порядок на множестве всех перестановок определяется следующим образом:

$P^1<P^2$ тогда и только тогда, когда существует такое $t\geq 1$, что $P^1[t]\leq P^2[t]$ и $P^1[i]=P^2[i]$ для всех $i<t$.

При решении данной задачи решается и задача получения следующей в лексикографическом порядке перестановки.

Антилексикографический порядок на множестве всех перестановок определяется следующим образом:

$P^1<P^2$ тогда и только тогда, когда существует такое $t\leq N$, что $P^1[t]>P^2[t]$ и $P^1[i]=P^2[i]$ для всех $i>t$.

Пример. N=3. В таблице 2.2 приведены все лексикографические и антилексикографические порядки на множестве всех перестановок для N=3.

Таблица 2.2

Лексикографический (A)	Антилексикографический (B)
123	123
132	213
213	132
231	312
312	231
321	321

Пример. N=4. В таблице 2.3 столбцы, обозначенные буквой A, означают генерацию перестановок в лексикографическом порядке, буквой B — антилексикографическом.

Таблица 2.3

A	B	A	B	A	B	A	B
1234	1234	2134	1243	3124	1342	4123	2341
1243	2134	2143	2143	3142	3142	4132	3241
1324	1324	2314	1423	3214	1432	4213	2431
1342	3124	2341	4123	3241	4132	4231	4231
1423	2314	2413	2413	3412	3412	4312	3421
1432	3214	2431	4213	3421	4312	4321	4321

Пример. Перестановка 11, 8, 5, 1, 7, 4, 10, 9, 6, 3, 2. Какая следующая перестановка идет в лексикографическом порядке?

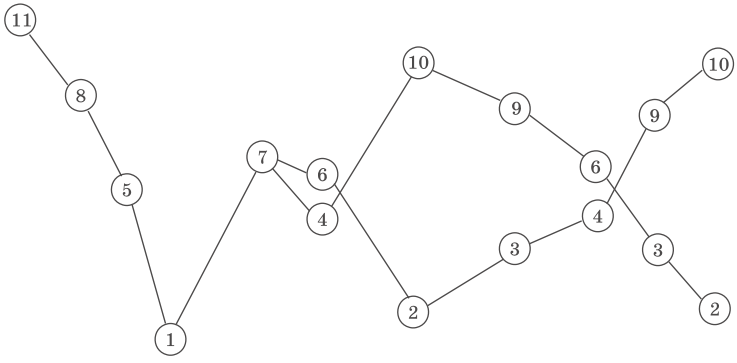


Рис. 2.2

Находим «скачок» — 4 меньше 10. Затем «в хвосте» перестановки находим первый элемент с конца перестановки, больший значения 4, на рисунке 2.2 это значение 6. Меняем местами элементы 4 и 6 и «хвост» перестановки ухудшаем максимально, т. е. расставляем элементы в порядке возрастания. Процедура получения следующей в лексикографическом порядке перестановки имеет вид:

```
Procedure GetNext;
 Var i,j:Integer;
 Begin {*Для перестановки N, N-1,…,1 процедура
        не работает, проверить.*}
  i:=N;
  While (i>1) And (P[i]<P[i-1]) Do Dec(i);
       {*Находим "скачок".*}
  j:=N;
  While P[j]<P[i-1] Do Dec(j);  {*Находим первый
                элемент, больший значения P[i-1].*}
  Swap(P[i-1],P[j]);
  For j:=0 To (N-i+1) Div 2 - 1 Do
    Swap(P[i+j],P[N-j]);{*Переставляем элементы "хвоста"
              перестановки в порядке возрастания.*}
 End;
Procedure Swap(Var a,b:Integer);
 Var t:Integer;
 Begin
  t:=a;a:=b;b:=t;
 End;
```

Фрагмент общей схемы генерации:

```
  ...
  Init;{*Инициализация: в массиве P - первая
       перестановка, в массиве Last - последняя.*}
  Print;{*Вывод перестановки.*}
  While Not(Eq(P,Last)) Do
    Begin{*Функция Eq
          определяет равенство текущей перестановки
          и последней; если не равны,
          то генерируется следующая*}
      GetNext;
      Print;
    End;
```

Изменим задачу. Необходимо перечислить все перестановки чисел от 1 до N так, чтобы каждая следующая перестановка получалась из предыдущей с помощью транспозиции двух соседних элементов.

Например: 321→231→213→123→132→312.

Возьмем перестановку $p_1, p_2, ..., p_N$ и сопоставим с ней следующую последовательность целых неотрицательных чисел $y_1, y_2, ..., y_N$. Для любого i от 1 до N найдем номер позиции s, в которой стоит значение i в перестановке, т. е. такое s, что $p_s=i$, и подсчитаем количество чисел, меньших i среди $p_1, p_2, ..., p_{s-1}$, это количество и будет значением y_i.

Пример:

Таблица 2.4

i	1	2	3	4	5	6	7	8	9	10	11
t_i	11	8	5	1	7	4	10	9	6	3	2
y_i	0	1	1	1	0	3	2	0	5	5	0

Очевидно, что $0 \le y_i < i$. Всего таких последовательностей $N!$, т. е. множества перестановок и последовательностей чисел совпадают по мощности. Построение последовательности чисел из перестановки очевидно, как и то, что разным перестановкам соответствуют разные последовательности чисел. Таблица 2.4 называется таблицей инверсий. Пару (p_i, p_j), такую, что $i<j$ и $p_i>p_j$, называют инверсией перестановки. Каждая инверсия как бы нарушает порядок в перестановке, единственная перестановка, не имеющая инверсий, — это (1, 2, 3, ..., N). М. Холл установил, что таблица инверсий единственным образом определяет соответствующую перестановку — обратное построение.

Пример. Последовательность 0111. Четверка записана на втором месте, ибо только одна цифра в перестановке «стоит» левее ее; имеем *4**. Тройка с учетом занятого места находится на третьем месте — *43*. Двойке с учетом занятых мест принадлежит четвертое место, т. е. получаем перестановку 1432.

Еще примеры. Дана последовательность чисел 0023. Четверка стоит на четвертом месте (3+1), тройка находится на третьем месте (2+1) — **34, двойка (0+1) — на первом месте — получаем 2134. Еще одна последовательность — 0112. Четверка на третьем месте (2+1), тройка на втором (1+1), двойка на втором (1+1), но с учетом занятых мест на четвертом получаем перестановку 1342. Идею осознали?

В таблице 2.5 приведены последовательности чисел и соответствующие перестановки для N=4.

Таблица 2.5

0000	4321	0011	2431	0123	1234	0112	1342
0001	3421	0010	4231	0122	1243	0113	1324
0002	3241	0020	4213	0121	1423	0103	3124
0003	3214	0021	2413	0120	4123	0102	3142
0013	2314	0022	2143	0110	4132	0101	3412
0012	2341	0023	2134	0111	1432	0100	4312

Зрительный образ предлагаемой схемы генерации. Пусть есть доска в форме лестницы (см. рис. 2.3). Высота i вертикали равна i. В первоначальном состоянии шашки стоят на первой горизонтали. Стрелки указывают направление движения шашек (вверх). За один ход можно передвинуть одну шашку в направлении стрелки. Если шашку передвинуть нельзя, то осуществляется переход к следующей слева шашке. После того как найдена шашка, которую можно передвинуть помимо передвижения шашки, осуществляется смена направления движения всех шашек справа от найденной. Ниже на рисунке 2.3 приведена смена состояний шашек на доске при N=4, начиная с последовательности чисел 0023. Под последовательностями чисел записаны соответствующие им перестановки.

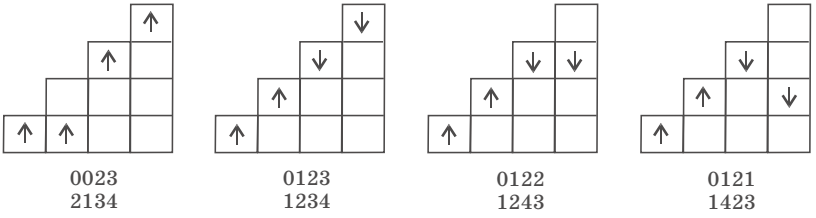


Рис. 2.3

Оказывается, что изменение на единицу одного из элементов последовательности соответствует транспозиции двух соседних элементов перестановки. Увеличение $y[i]$ на единицу соответствует транспозиции числа i с правым соседом, уменьшение — с левым соседом. Итак, помимо генерации последовательностей необходимо уметь находить число i в перестановке и менять его местами с одним из «соседей», но это уже чисто техническая задача.

Опишем используемые данные.

```
Const Nmax=12;
Var     P:Array[1..Nmax] Of 0..Nmax;
{*Хранение перестановки.*}
        Y:Array[1..Nmax] Of 0..Nmax-1;
{*Хранение последовательности чисел.*}
        D:Array[1..Nmax] Of -1..1;
{*Направление движения "шашек".*}
```

Процедура формирования начальной перестановки и начальных значений в массивах Y и D.

```
Procedure First;
  Var  i:Integer;
  Begin
   For i:=1 To N Do
     Begin
        P[i]:=N-i+1; Y[i]:=0;D[i]:=1;
     End;
  End;
```

Поиск номера «шашки», которую можно передвинуть.

При $D[i]=1$ и $Y[i]=i-1$ или $D[i]=-1$ и $Y[i]=0$ шашку с номером i нельзя передвинуть.

```
Function Ok:Integer;{*Если значение функции равно
            1, то нет ни одной шашки, которую можно
            передвинуть.*}
  Var i:Integer;
   Begin
   i:=N;
   While (i>1) And (((D[i]=1) And (Y[i]=i-1)) Or
         ((D[i]=-1) And (Y[i]=0))) Do Dec(i);
   Ok:=i;
  End;
```

Основной алгоритм генерации перестановок имеет вид:

```
Begin
 First;
 pp:=True;
 While pp Do
   Begin
     Solve(pp);
     If pp Then Print;
   End;
End;
```

Уточним процедуру генерации следующей перестановки.

```
Procedure Solve(Var q:Boolean);
 Var i,j,dj:Integer;
 Begin
  i:=Ok; q:=(i>1);{*Находим номер шашки, которую
                   можно передвинуть.*}
  If i>1 Then
    Begin
     Y[i]:=Y[i]+D[i];{*Передвигаем шашку - изменяем
                      последовательность чисел.*}
     For j:=i+1 To N Do D[j]:=-D[j];{*Изменяем
           направление движения шашек, находящихся
           справа от передвинутой.*}
     j:=Who_i(i);{*Находим позицию в перестановке,
           в которой записано число i.*}
     dj:=j+D[i];{*Определяем соседний элемент
        перестановки для выполнения транспозиции.*}
     Swap(P[j],P[dj]);{*Транспозиция.*}
    End;
 End;
```

Остался последний фрагмент алгоритма, требующий уточнения, — поиск в перестановке позиции, в которой находится элемент *i*. Он достаточно прост.

```
Function Who_i(i:Integer):Integer;
 Var j:Integer;
 Begin
  j:=N;
  While (j>0) And (P[j]<>i) Do Dec(j);
  Who_i:=j;
 End;
```

Четвертая задача. По номеру определить перестановку относительно порядка, который введен на множестве перестановок.

Остановимся на лексикографическом порядке. Рассмотрим идею решения на примере.

Пусть N равно 8 и дан номер L, равный 37021. Найдем соответствующую перестановку. Пусть на первом месте записана единица. Таких перестановок 7!, или 5040 (1*******). При 2 тоже 5040 (2*******). Итак, 37021 *Div* 5040=7. Следовательно, первая цифра в перестановке 8. Новое значение L (37021 *Mod* 5040=1741) 1741. Продолжим рассуждения и оформим их в виде таблицы 2.6.

Таблица 2.6

i	*L*	$P_{старая}$	*P*	*L Div P*	*L Mod P*	$P_{новая}$
1	37021	********	7!=5040	7	1741	8*******
2	1741	8*******	6!=720	2	301	83******
3	301	83******	5!=120	2	61	834*****
4	61	834*****	4!=24	2	13	8345****
5	13	8345****	3!=6	2	1	83456***
6	1	83456***	2!=4	0	1	834561**
7	1	834561**	1!=1	1	0	8345617*
8	0	8345617*	0!=1			83456172

Обратим внимание на третью строку, в которой на третье место записывается цифра 4. То, что записываются не цифры 1 и 2, очевидно: их требуется пропустить. Цифра три «занята», поэтому записываем 4. Точно так же в строках 4, 5 и 7.

О программной реализации. Пусть $N \le 12$, таким образом, значение *L* не превосходит максимального значения типа *LongInt*. Определим следующие константы.

```
Const Nsmall=12;
    ResSm:Array[0..Nsmall] Of LongInt
=(1,1,2,6,24,120,720,5040,40320,362880,3628800,39916
800,479001600); {*Значения N! для N от 0 до 12.*}
```

И процедура.

```
Procedure GetPByNum(L:LongInt);
 Var  Ws:Set Of Byte;
      i,j,sc:Integer;
 Begin
  Ws:=[];{*Множество для фиксации цифр,
           задействованных в перестановке.*}
  For i:=1 To N Do
    Begin
     Sc:=L Div ResSm[N-i];L:=L Mod ResSm[N-i];
     j:=1;
     While (sc<>0) Or (j In Ws) Do
      Begin
        If Not (j In Ws) Then Dec(sc);
        Inc(j);
      End;
     Ws:=Ws+[j];{*Нашли очередную цифру.*}
```

```
      P[i]:=j;
    End;
  End;
```

Для более глубокого понимания сути приведем часть трассировки работы процедуры для ранее рассмотренного примера. Результаты трассировки приведены в таблице 2.7.

Таблица 2.7

I	*Sc*	*L*	*J*	*Ws*	*P*
		37021			********
1	7	1741	1	[]	
	6		2		
	5		3		
	4		4		
	3		5		
	2		6		
	1		7		
	0		8	[8]	8********
2	2	301	1		
	1		2		
	0		3	[3,8]	83******
3	2	61	1		
	1		2		
	1		3		
	0		4	[3,4,8]	834******
4	2	13	1		
	1		2		
	1		3		
	1		4		
	0		5	[3,4,5,8]	8345****
...	...	...	...	...	...

Пятая задача. По перестановке получить ее номер, не выполняя генерацию множества перестановок.

Начнем с примера. Пусть N равно 8 и дана перестановка 53871462.

Схема:

7!* <количество цифр в перестановке на 1-м месте, идущих до цифры 5, с учетом занятых цифр — ответ 4>

6!* < количество цифр в перестановке на 2-м месте, идущих до цифры 3, с учетом занятых цифр — ответ 2>

5!× < количество цифр в перестановке на 3-м месте, идущих до цифры 8, с учетом занятых цифр — ответ 5>
4!× < количество цифр в перестановке на 4-м месте, идущих до цифры 7, с учетом занятых цифр — ответ 4>
3!× < количество цифр в перестановке на 5-м месте, идущих до цифры 1, с учетом занятых цифр — ответ 0>
2!× < количество цифр в перестановке на 6-м месте, идущих до цифры 4, с учетом занятых цифр — ответ 1>
1!× < количество цифр в перестановке на 7-м месте, идущих до цифры 6, с учетом занятых цифр — ответ 1>
Итак,
7!×4+6!×2+5!×5+4!×4+3!×0+2!×1+1!×1=4×5040+2×720+5×120+4×
×24+0×6+1×2+1×1=22305.

```
Function GetNumByP:LongInt;
 Var  ws:Set Of Byte;
      i,j,sq:Integer;sc:LongInt;
 Begin
  ws:=[];{*Множество цифр перестановки.*}
  sc:=1; {*Номер перестановки.*}
  For i:=1 To N Do
   Begin
      j:=1;sq:=0;{*Определяем количество цифр.*}
      While j<P[i] Do
        Begin
          If Not(j In ws) Then Inc(sq);
          Inc(j);
        End;
      ws:=ws+[P[i]];{*Цифра P[i] задействована.*}
      sc:=sc+ResSm[N-i]*sq;{*Умножаем число
         перестановок на значение текущей цифры.*}
   End;
  GetNumByP:=sc;
End;
```

2.2.2. Размещения

Первая задача. Подсчитать количество размещений для небольших значений N и M можно с помощью следующих простых функций.

```
Function Plac(n,m:LongInt):LongInt;
 Var i,z:LongInt;
 Begin
  z:=1;
```

```
 For i:=0 To m-1 Do z:=z*(n-i);
 Plac:=z;
End;
```

Рекурсивный вариант реализации.

```
Function Plac(n,m:LongInt):LongInt;
 Begin
  If m=0
   Then Plac:=1
   Else Plac:=(n-m+1)*Plac(n,m-1);
 End;
```

Функции дают правильный результат при $N \le 12$. Для больших значений N необходимо использовать «длинную» арифметику.

Вторая задача. Сгенерировать все размещения для заданных значений N и M в лексикографическом порядке.

Пример. $N=4$, $M=3$. В таблице 2.8 приведены размещения в лексикографическом порядке.

Таблица 2.8

123	142	231	312	341	421
124	143	234	314	342	423
132	213	241	321	412	431
134	214	243	324	413	432

Текст решения имеет вид:

```
Program GPlac;
Const n=4; m=3;{*Значения n и m взяты в качестве
                 примера.*}
 Var    A:Array[1..m] Of Integer;{*Массив для
                 хранения элементов размещения.*}
 S:Set Of Byte;{*Для хранения использованных
                в размещении цифр.*}
  Procedure Solve(t:Integer);{*Параметр t
          определяет номер позиции в размещении.*}
   Var i:Byte;
   Begin
    For i:=1 To n Do{*Перебираем цифры и находим
                     1-ю неиспользованную.*}
     If Not(i In S) Then
      Begin
```

```
      S:=S+[i];{*Запоминаем ее в множестве занятых
               цифр.*}
      A[t]:=i;{*Записываем ее в размещение.*}
      If t<m Then Solve(t+1) Else Print;{*Если
                  размещение не получено, то идем
                  к следующей позиции, иначе
                  выводим очередное размещение.*}
      S:=S-[i];{*Возвращаем цифру в число
                  неиспользованных.*}
    End;
  End;
```

Фрагмент основной программы.

```
Begin
 S:=[];  Solve(1);
End.
```

Для генерации размещений можно воспользоваться процедурой генерации перестановок из предыдущего пункта. Требуется только уметь «вырезать» из очередной перестановки первые M позиций и исключать при этом повторяющиеся последовательности.

Третья задача. По заданному размещению найти следующее за ним в лексикографическом порядке.

Свободными элементами размещения назовем те элементы из множества от 1 до N, которых нет в текущем размещении (последовательности из M элементов).

Пример. N=5, M=3 и размещение 1 3 4. Свободными элементами являются 2 и 5. Первый вариант решения заключается в том, чтобы дописать в размещение свободные элементы в убывающем порядке (1 3 4 5 2), сгенерировать следующую перестановку (1 3 5 2 4) и отсечь первые M элементов (1 3 5). Получаем следующее размещение. Реализация этой же идеи, но без обращения к генерации следующей перестановки приводится ниже по тексту.

Ее суть:

- Начинаем просмотр размещения с последнего элемента M и идем, если необходимо, до первого.
- Находим первый свободный элемент, который больше, чем элемент в рассматриваемой позиции размещения.
- Если такой элемент найден, то заменяем им текущий, а в «хвост» размещения записываем свободные элементы в порядке возрастания и заканчиваем работу.

- Если такого элемента не найдено, то переходим к следующей позиции.

```
Const n=4; m=3;
 Var A:Array[1..m] Of Byte;
Procedure GetNext;
  Var i,j,k,q:Integer;
      Free:Array[1..n] Of Boolean;{*Массив для
           хранения признаков занятости элементов
           в размещении.*}
  Function FreeNext(t:Byte):Byte;{*Функция поиска
           первого не занятого элемента.*}
   Begin
    While  (t<=n) And Not(Free[t]) Do Inc(t);
          {*Находим первый свободный элемент.*}
    If t>n
     Then FreeNext:=0{*Если такого элемента
             нет, то значение функции равно нулю.*}
     Else FreeNext:=t;{*Номер первого свободного
          элемента.*}
   End;
  Begin
   For i:=1 To n Do Free[i]:=True;{*Массив
          свободных элементов.*}
   For i:=1 To m Do Free[A[i]]:=False;{*По
          размещению исключаем занятые элементы.*}
   i:=m;{*Начинаем с конца размещения.
          Предполагаем, что анализируемое
          размещение не является последним.*}
   While (i>0) And (FreeNext(A[i])=0) Do
    Begin
          {*Пока не нашли позицию в размещении,
           элемент в которой допускается изменить,
           выполняем действия из цикла. При нашем
           предположении такая позиция обязательно
           существует.*}
     Free[A[i]]:=True;{*Освобождаем элемент,
          записанный в позиции i.*}
     Dec(i);{*Переходим к следующей позиции.*}
    End;
   Free[A[i]]:=True;{*Переводим текущий элемент
          в найденной позиции в свободные.*}
   q:=FreeNext(A[i]+1);{*Находим свободный
          элемент, больший текущего.*}
```

```
    Free[q]:=False;{*Считаем его занятым.*}
    A[i]:=q;{*Записываем его в размещение.*}
    k:=1;{*Формируем "хвост" размещения.*}
    For j:=i+1 To m Do
      Begin{*Со следующей позиции
            до конца размещения.*}
        While (k<=n) And Not(Free[k]) Do{*Пока не
              найдем первый свободный элемент.*}
         If k>=n Then k:=1 Else Inc(k);
        A[j]:=k;{*Записываем найденный элемент
              в размещение.*}
        Free[k]:=False;{*Считаем его занятым.*}
      End;
  End;
```

Четвертая задача. При заданных значениях N и M по номеру размещения L определить соответствующее размещение (упорядочены в лексикографическом порядке).

Рассмотрим размещения из 5 по 3 элемента, их 60. В таблице 2.9 часть размещений, вместе с их номерами, приводится в лексикографическом порядке. Первому размещению соответствует номер 0.

Таблица 2.9

0	123	6	142	12	213	18	241	24	312	30	341
1	124	7	143	13	214	19	243	25	314	31	342
2	125	8	145	14	215	20	245	26	315	32	345
3	132	9	152	15	231	21	251	27	321	33	351
4	134	10	153	16	234	22	253	28	324	34	352
5	135	11	154	17	235	23	254	29	325	...	

Значение A_4^2 (в общем случае A_{N-1}^{M-1}) равно 12. Количество размещений с фиксированным значением в первой позиции равно 12, а это уже путь к решению.

Пусть нам задан номер 32 и массив свободных элементов 1 2 3 4 5 (номера элементов массива считаются с 0). Вычисляя 32 *Div* 12, получаем 2, следовательно, в первой позиции записана цифра, стоящая на 2-м месте в массиве свободных элементов, а это цифра 3. Оставшееся количество номеров 32 *Mod* 12 =8, массив свободных элементов 1 2 4 5. Продолжим процесс вычисления — 8 *Div* 3=2 (A_3^1 =3), следующая цифра 4. Оставшееся количество номеров 8 *Mod* 3 =2, массив свободных номеров 1 2 5. Продолжим — 2 *Div* 1 =2, что соответствует цифре 5.

```
Const n=5;m=3;
 Var A:Array[1..m] Of Integer;
    L:LongInt;{*Номер размещения.*}
  Procedure GetPByNum(L:LongInt);{*В процедуре
            использована функция Plac - вычисления
            количества размещений.*}
   Var i,j,q,t:LongInt;
      Free:Array[1..n] Of Byte;{*Массив свободных
            элементов размещения.*}
   Begin
    For i:=1 To n Do Free[i]:=i;{*Начальное
              формирование.*}
    For i:=1 To m Do
     Begin{*i - номер позиции
              в размещении.*}
        t:=Plac(n-i,m-i);{*Количество размещений,
              приходящихся на один фиксированный
              элемент в позиции i.*}
        q:=L Div t;{*Вычисляем номер свободного
              элемента размещения.*}
        A[i]:=Free[q+1];{*Формируем элемент
              размещения.*}
        For j:=q+1 To n-i Do Free[j]:=Free[j+1];
             {*Сжатие, найденный элемент
             размещения исключаем из свободных.*}
        L:=L Mod t;{*Изменяем значение номера
              размещения (переход к меньшей
               размерности).*}
    End;
   End;
```

Пятая задача. По размещению определить его номер (размещения упорядочены в лексикографическом порядке).

Поясним идею решения на примере. Дано размещение 345. Количество свободных элементов, меньших 3, равно 2 (12×2=24). Количество свободных номеров, меньших 4, также 2 — 24+3×2=30. И наконец, меньших пяти, также 2.

Ответ: 12×2+3×2+1×2=32.

```
  Function GetNumByP:LongInt;{*Используется
             процедура Plac вычисления количества
             размещений.*}
   Var L,i,j,num:LongInt;
      ws:Set Of Byte;{*Множество элементов
             размещения.*}
```

```
Begin
 ws:=[];
 L:=0;
 For i:=1 To m Do
  Begin{*i - номер позиции
           размещения.*}
      num:=0;{*Счетчик числа незанятых элементов.*}
      For j:=1 To A[i]-1 Do
       If Not(j In ws) Then Inc(num);{*Если элемента
          j нет в ws, то увеличиваем значение
          счетчика числа незанятых элементов,
        которые встречаются до значения A[i]. *}
      ws:=ws+[A[i]];{*Элемент A[i] задействован
                      в размещении.*}
      L:=L+num*Plac(n-i,m-i);{*Значение счетчика
          умножаем на количество размещений,
          приходящихся на одно значение
          в позиции с номером i.*}
  End;
 GetNumByP:=L;
End;
```

2.2.3. Сочетания

Между k-элементными подмножествами N-элементного множества и возрастающими последовательностями длины k с элементами из множества целых чисел от 1 до N (сочетаниями) существует взаимно однозначное соответствие. Так, например, подмножеству [2, 6, 1, 3] соответствует последовательность чисел 1, 2, 3, 6. Таким образом, решая задачи подсчета и генерации всех сочетаний, мы решаем аналогичные задачи для k-элементных подмножеств. Лексикографический порядок на множестве последовательностей определяется так же, как и для перестановок.

Пример. В таблице 2.10 приведен лексикографический порядок для C_8^4.

Таблица 2.10

1234	1246	1356	2356
1235	1256	1456	2456
1236	1345	2345	3456
1245	1346	2346	

Первая задача. Подсчитать количество сочетаний для данных значений N и k.

Использование формулы $C_N^k = \frac{N!}{k! \times (N-k)!}$ явно не продуктивно. Факториал представляет собой быстровозрастающую функцию, поэтому при вычислении числителя и знаменателя дроби может возникнуть переполнение, хотя результат — число сочетаний — не превышает, например, значения *MaxInt*. Воспользуемся для подсчета числа сочетаний формулой

$$C_N^k = C_{N-1}^{k-1} + C_{N-1}^k, \text{ где } N>1,\ 0<k<N.$$

Получаем знаменитый треугольник Паскаля (см. таблицу 2.11). Просматривается явная схема вычислений. Очевидно, что если в задаче не требуется постоянно использовать значения C_N^k для различных значений N и k, а требуется только подсчитать одно значение C_N^k, то хранить двумерный массив в памяти компьютера нет необходимости.

Таблица 2.11

N \ k	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0	0	0
3	1	3	3	1	0	0	0	0	0	0
4	1	4	6	4	1	0	0	0	0	0
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1	0	0	0
7	1	7	21	35	35	21	7	1	0	0
8	1	8	28	56	70	56	28	8	1	
9	1	9	36	84	126	126	84	36	9	1

Приведем процедуры вычисления C_N^k для того и другого случая. Подсчет всех значений C_N^k.

```
Const MaxN=100;
Var SmallSc:Array[0..MaxN,0..MaxN] Of LongInt;
    {*Нулевой столбец и нулевая строка необходимы,
     это позволяет обойтись без анализа на выход
     за пределы массива.*}
Procedure FillSmallSc;
 Var i,j:Integer;
```

```
 Begin
  FillChar(SmallSc,SizeOf(SmallSc),0);
  For i:=0 To N Do SmallSc[i,0]:=1;
  For i:=1 To N Do
   For j:=1 To k Do
    If SmallSc[i-1,j]*1.0+SmallSc[i-1,j-1]>
       MaxLongInt
     Then SmallSc[i,j]:=MaxLongInt
     Else
SmallSc[i,j]:=SmallSc[i-1,j]+SmallSc
        [i-1,j-1]; {*Умножение на 1.0 переводит
        целое число в вещественное, поэтому
        переполнения при сложении не происходит.
        Стандартный прием, обязательный при "игре"
        на границах диапазона целых чисел.*}
  End;
```

Второй вариант реализации процедуры.

```
Type SmallSc=Array[0..MaxN] Of LongInt;
Function Res(k:Integer):LongInt;
 Var A,B:SmallSc;
     i,j:Integer;
 Begin
  FillChar(A,SizeOf(A),0);
  FillChar(B,SizeOf(B),0);
  A[0]:=1;A[1]:=1;
  For i:=1 To N Do
   Begin
     B[0]:=1;B[1]:=1;
     For j:=1 To k Do B[j]:=A[j]+A[j-1];{*Если
       число больше максимального значения типа
       LongInt,то необходимо работать с "длинной"
       арифметикой. Данная операция должна быть
       изменена - вызов процедуры сложения двух
       "длинных" чисел.*}
     A:=B;
   End;
  Res:=A[k];
 End;
```

Вторая задача. Сгенерировать все сочетания в лексикографическом порядке.

Рассмотрим пример для N=7 и k=5. Число сочетаний равно 21 (см. таблицу 2.12).

Таблица 2.12

0	12345	7	12457	14	14567
1	12346	8	12467	15	23456
2	12347	9	12567	16	23457
3	12356	10	13456	17	23467
4	12357	11	13457	18	23567
5	12367	12	13467	19	24567
6	12456	13	13567	20	34567

Начальное сочетание равно 1, 2, ..., k, последнее — $N-k+1$, $N-k+2$, ..., N. Переход от текущего сочетания к другому осуществляется по следующему алгоритму. Просматриваем сочетание справа налево и находим первый элемент, который можно увеличить. Увеличиваем этот элемент на единицу, а часть сочетания (от этого элемента до конца) формируем из чисел натурального ряда, следующих за ним.

```
Procedure GetNext;{*Предполагается, что текущее
      сочетание (хранится в массиве C) не является
      последним.*}
 Var i,j:Integer;
 Begin
  i:=k;
  While (C[i]+k-i+1>N) Do Dec(i); {*Находим
      элемент, который можно увеличить.*}
  Inc(C[i]);{*Увеличиваем на единицу.*}
  For j:=i+1 To k Do C[j]:=C[j-1]+1;{*Изменяем
     стоящие справа элементы.*}
 End;
```

Третья задача. По номеру L определить соответствующее сочетание.

Для решения этой задачи необходимо иметь (вычислить) массив *SmallSc*, т. е. использовать первую схему вычисления числа сочетаний (первая задача) для данных значений N и k. Порядок на множестве сочетаний лексикографический. В таблице 2.12 номера с 0 по 14 имеют сочетания, в которых на первом месте записана единица. Число таких сочетаний — C_6^4 («израсходован» один элемент из N и один элемент из k). Сравниваем число L со значением C_6^4. Если значение L больше C_6^4, то на первом месте в сочетании записана не единица, а большее число. Вычтем из L значение C_6^4 и продолжим сравнение.

```
Procedure GetWhByNum(L:LongInt);
 Var i,j,sc,ls:Integer;
 Begin
  sc:=n;
  ls:=0;{*Цифра сочетания.*}
  For i:=1 To k Do
  Begin{*i - номер элемента в сочетании;
        k-i - число элементов в сочетании.*}
    j:=1; {*sc-j - число элементов (n), из которых
          формируется сочетание.*}
     While L-SmallSc[sc-j,k-i]>=0 Do
    Begin{*Для данной позиции в сочетании и числа
          элементов в сочетании находим тот элемент
          массива SmallSc, который превышает
          текущее значение L.*}
       Dec(L,SmallSc[sc-j,k-i]);
       Inc(j);{*Невыполнение условия цикла While
          говорит о том, что мы нашли строку
          таблицы SmallSc, т.е. то количество
          элементов, из которых формируется
          очередное сочетание, или тот интервал
          номеров сочетаний (относительно
          предыдущей цифры), в котором находится
          текущее значение номера L.*}
    End;
    C[i]:=ls+j;{*Предыдущая цифра плюс приращение.*}
    Inc(ls,j);{*Изменяем значение текущей цифры
           сочетания.*}
    Dec(sc,j);{*Изменяем количество элементов,
           из которых формируется остаток
           сочетания.*}
  End;
 End;
```

В таблицах 2.13, 2.14 и 2.15 приведены результаты ручной трассировки процедур *GetWhByNum* для различных N, k и L.

Таблица 2.13

N=7, k=5, L=17.

i	*j*	*L*	*sc*	*ls*	*SmallSc[sc-j,k-i]*	*C*	$sc_{новое}$	$ls_{новое}$
1	1	17	7	0	15	*****	-	-
	2	2	7	0	5	2****	5	2
2	1	2	5	2	5	23***	4	3

3	1	2	4	3	3	234**	3	4
4	1	2	3	4	2	234**		
	2	0	3	4	1	2346*	1	6
5	1	0	1	6	1	23467	0	7

Таблица 2.14

N=7, k=5, L=0.

i	*j*	*L*	*sc*	*ls*	*SmallSc[sc-j,k-i]*	*C*	$sc_{новое}$	$ls_{новое}$
1	1	0	7	0	15	1****	6	1
2	1	0	6	1	10	12***	5	2
3	1	0	5	2	6	123**	4	3
4	1	0	4	3	3	1234*	3	4
5	1	0	3	4	1	12345	2	5

Таблица 2.15

N=7, k=5, L=20.

i	*j*	*L*	*sc*	*ls*	*SmallSc[sc-j,k-i]*	*C*	$sc_{новое}$	$ls_{новое}$
1	1	20	7	0	15	*****	-	-
	2	5	7	0	5	*****	-	-
	3	0	7	0	1	3****	4	3
2	1	0	4	3	1	34***	3	4
3	1	0	3	4	1	345**	2	5
4	1	0	2	5	1	3456*	1	6
5	1	0	1	6	1	34567	0	7

Для полного понимания алгоритма выполните ручную трассировку при N=9, k=5, L=78. Ответ — 23479. Однако достаточно, пора и честь знать.

Четвертая задача. По сочетанию получить его номер.

Как и в предыдущей задаче, необходимо иметь массив *SmallSc*. Эта задача проще предыдущей. Требуется аккуратно просмотреть часть массива *SmallSc*. Эта часть массива определяется сочетанием, для которого ищется номер. Например, на первом месте в сочетании стоит цифра 3 (N=7, k=4). Ищется сумма $C_7^3 + C_6^3 + C_5^3$ — количество сочетаний, в которых на первом месте записана цифра 1, цифра 2 и цифра 3. Аналогично для следующих цифр сочетания. Предполагаем, что $C[0]$ равно 0 — технический прием, позволяющий избежать лишних проверок условий.

```
Function GetNumByWh:LongInt;
 Var sc:LongInt;
    i,j:Integer;
 Begin
  sc:=1;
  For i:=1 To k Do
   For j:=C[i-1]+1 To C[i]-1 Do Inc(sc,SmallSc[N-j,k-i]);
  GetNumByWh:=sc;
 End;
```

В таблицах 2.16 и 2.17 приведены результаты ручной трассировки функции *GetNumByWh* для различных N, k и L.

Таблица 2.16

N=9, k=5, C=023479.

i	j	$sc_{старое}$	SmallSc[N-j,k-i]	$sc_{новое}$
1	1	1	70	71
2	от 3 до 2, цикл по j не работает	71	-	-
3	от 4 до 3, цикл по j не работает	71	-	-
4	5	71	4	75
	6	75	3	78
5	8	78	1	79

Таблица 2.17

N=7, k=5, C=023467.

i	j	$sc_{старое}$	SmallSc[N-j,k-i]	$sc_{новое}$
1	1	1	15	16
2	от 3 до 2, цикл по j не работает	16	-	-
3	от 4 до 3, цикл по j не работает	16	-	-
4	5	16	2	18
5	от 7 до 6, цикл по j не работает	18	-	-

Обратите внимание на то, что нумерация сочетаний в задачах 3 и 4 различна. В задаче 3 сочетания нумеруются с 0, а в задаче 4 — с 1.

Пятая задача. Выберем для представления сочетания другой вид — последовательность из 0 и 1. Последовательность храним в массиве C из N элементов со значениями 0 или 1, при этом $C[i]$=1 говорит о том, что элемент $(N-i+1)$ есть в сочетании (подмножестве).

Пусть N=7, k=5. В таблице 2.18 представлены последовательности в той очередности, в которой они генерируются, и соответствующие им сочетания.

Таблица 2.18

0	0011111	12345	7	1010111	12357	14	1101110	23467
1	0101111	12346	8	1011011	12457	15	1110011	12567
2	0110111	12356	9	1011101	13457	16	1110101	13567
3	0111011	12456	10	1011110	23457	17	1110110	23567
4	0111101	13456	11	1100111	12367	18	1111001	14567
5	0111110	23456	12	1101011	12467	19	1111010	24567
6	1001111	12347	13	1101101	13467	20	1111100	34567

Принцип генерации последовательностей.

Начинаем с конца последовательности. Находим пару $C[i]$=0 и $C[i+1]$=1. Если такой пары нет, то получена последняя последовательность. Если есть, то заменяем на месте i нуль на единицу и корректируем последовательность справа от i. Последовательность должна быть минимальна с точки зрения введенного порядка, поэтому вначале записываются 0, а затем 1, при этом количество 1 в последовательности должно быть сохранено. Опишем основные фрагменты реализации этого алгоритма.

```
Const MaxN=…;
Type MyArray=Array[1..MaxN] Of 0..1;
Var Curr, Last:MyArray;{*Текущая и последняя
        последовательности.*}
        i,j,Num,N,k:Integer;
Begin
  ...
  For i:=1 To N Do
   Begin Curr[i]:=0;Last[i]:=0;End;
  For i:=1 To k Do
   Begin Curr[N-i+1]:=1;Last[i]:=1; End;
      {*Формируем начальную
        и последнюю последовательности.*}
  While Not Eq(Curr,Last) Do
   Begin
     i:=n-1;
     While (i>0) And Not((Curr[i]=0) And
          (Curr[i+1]=1)) Do Dec(i);
          {*Находим пару 0 - 1, она есть, так как
          последовательность не последняя.*}
```

```
    Num:=0;
    For j:=i+1 To n Do Inc(Num,Curr[j]);
          {*Подсчитываем количество единиц.*}
    Curr[i]:=1;
    For j:=i+1 To n-Num+1 Do Curr[j]:=0;{*Записываем
        нули.*}
    For j:=n-Num+2 To n Do Curr[j]:=1;{*Записываем
        единицы.*}
    …
  End;
End;
```

2.2.4. Разбиение числа на слагаемые

Дано натуральное число N. Его можно записать в виде суммы натуральных слагаемых:

$N=a_1+a_2+...+a_k$, где k, a_1, ..., a_k больше нуля.

Будем считать суммы эквивалентными, если они отличаются только порядком слагаемых. Класс эквивалентных сумм приведенного вида однозначно представляется последовательностями b_1, ..., b_k, упорядоченными по невозрастанию. Каждую такую последовательность b_1, ..., b_k назовем разбиением числа N на k слагаемых.

Как обычно, решаем четыре задачи: подсчет количества различных разбиений числа; генерация разбиений относительно определенного отношения порядка, введенного на множестве разбиений; получение разбиения по его номеру и задача, обратная третьей.

Первая задача. Приведем пример для N=8 (см. таблицу 2.19), а затем рассуждения, позволяющие написать программный код.

Таблица 2.19

1	11111111	9	3311	17	521
2	2111111	10	332	18	53
3	221111	11	41111	19	611
4	22211	12	4211	20	62
5	2222	13	422	21	71
6	311111	14	431	22	8
7	32111	15	44		
8	3221	16	5111		

Число разбиений числа N на k слагаемых обозначим через $P(N,k)$, общее число разбиений — $P(N)$. Очевидно, что значение $P(N)$ равно сумме $P(N,k)$ по значению k. Считаем, что $P(0)=P(1)=1$. Известен следующий факт.

Теорема. Число разбиений числа N на k слагаемых равно числу разбиений числа N с наибольшим слагаемым, равным k.

Теорема доказывается простыми рассуждениями на основе диаграмм Феррерса. Пример такой диаграммы приведен на рисунке 2.4.

15=6+4+3+1+1 15=5+3+3+2+1+1

Рис. 2.4

Главное, что следует из этого факта, — реальный путь подсчета числа разбиений. Число разбиений $P(i,j)$ равно сумме $P(i-j,k)$ по значению k, где k изменяется от 0 до j. Другими словами, j «выбрасывается» из i и подсчитывается число способов разбиения числа $i-j$ на k слагаемых. Проверьте этот алгоритм с помощью таблицы 2.20.

Таблица 2.20

	0	1	2	3	4	5	6	7	8	
1	0	1	0	0	0	0	0	0	0	1
2	0	1	1	0	0	0	0	0	0	2
3	0	1	1	1	0	0	0	0	0	3
4	0	1	2	1	1	0	0	0	0	5
5	0	1	2	2	1	1	0	0	0	7
6	0	1	3	3	2	1	1	0	0	11
7	0	1	3	4	3	2	1	1	0	15
8	0	1	4	5	5	3	2	1	1	22

Например, $P(8,3)=P(5,0)+P(5,1)+P(5,2)+P(5,3)$.
Определим данные.

```
Const MaxN=100;
Var N:LongInt;
    SmallSc:Array[0..MaxN,0..MaxN] Of LongInt;
Procedure FillSmallSc;
 Var i,j,k:Integer;
 Begin
  FillChar(SmallSc,SizeOf(SmallSc),0);
  SmallSc[0,0]:=1;
  For i:=1 To N Do
   For j:=1 To i Do
    For k:=0 To j Do
     Inc(SmallSc[i,j], SmallSc[i-j,k]);
  SmallSc[0,0]:=0;
 End;
```

Таблица значений сформирована, она нам потребуется для решения других задач. Для подсчета же числа разбиений осталось написать простую процедуру.

```
Procedure Calc;
 Var  i:Integer;
      Res:LongInt;
 Begin
  Res:=SmallSc[N,1];
  For i:=2 To N Do Res:=Res+SmallSc[N,i];
  WriteLn(Res);
 End;
```

Вторая задача. Генерация разбиений.

Во-первых, необходима структура данных для хранения разбиения. По традиции используем массив:

```
Var Now:Array[0..MaxN] Of Integer;
```

В *Now*[0] будем хранить длину разбиения, т. е. число задействованных элементов массива *Now*.

Возьмем из таблицы 2.19 одно из разбиений числа 8, например «41111», следующее «4211». В каком случае можно увеличить *Now*[2]? Видим, что *Now*[1] должно быть больше *Now*[2], т. е. в текущем разбиении находится «скачок» — *Now*[*i*–1]>*Now*[*i*], *Now*[*i*] увеличивается на единицу, а все следующие элементы разбиения берутся минимально возможными. Всегда ли возможна данная схема изменения разбиения? Например, разбиение «44», следующее — «5111». Таким образом, или «скачок», или *i* равно 1. Кроме того, изменяемый эле-

мент не может быть последним, ибо увеличение без уменьшения невозможно.

```
Procedure GetNext;
 Var i,j,sc:Integer;
 Begin
  i:=Now[0]-1;{*Предпоследний элемент разбиения.*}
  sc:=Now[i+1]-1;Now[i+1]:=0;{*В sc накапливаем
        сумму, это число представляется затем
        минимально возможными элементами.*}
  While (i>1) And (Now[i]=Now[i-1]) Do
   Begin
      sc:=sc+Now[i];Now[i]:=0;Dec(i);
      {*Находим "скачок".*}
   End;
  Inc(Now[i]);{*Увеличиваем найденный элемент
        на единицу.*}
  Now[0]:=i+sc;{*Изменяем длину разбиения.*}
  For j:=1 To sc Do Now[j+i]:=1;{*Записываем
        минимально возможные элементы,
        т. е. единицы.*}
 End;
```

А будет ли работать данный алгоритм для последнего разбиения (в нашем примере для «8»)? Оказывается, что нет. Проверьте.

Считаем, что в *Now* хранится первое разбиение, а в массиве *Last* — последнее. Они формируются, например, в процедуре типа *Init*. Схема генерации:

```
…
While Not(Eq(Now,Last)) Do
Begin{*Фунция Eq дает
        "истину", если текущее распределение
        совпадает с последним, и "ложь"
        в противном случае, можно сделать проще -
        While Now[0]<>1 Do ….*}
 GetNext;
 Print;{*Вывод разбиения.*}
End;
….
```

Третья задача. По номеру разбиения (*L*) получить само разбиение *Now*.

Нумерация разбиений осуществляется относительно введенного отношения порядка. Пусть L равно 0. Сделаем «набросок» алгоритма.

```
...
sc:=N;i:=1;{*sc - число, которое представлено
      разбиением, i - номер позиции в разбиении.*}
While sc<>0 Do
 Begin{*Пока число не исчерпано.*}
  j:=1;{*Число, которое должно записываться
        на место i в разбиении.*}
  ??????{*Значение j определяется по значению
        L, пока не знаем как.*}
  Now[i]:=j;sc:=sc-j;Inc(i);{*Формируем текущую
       позицию разбиения и готовимся к переходу для
       формирование новой.*}
 End;
...
```

Как это ни странно, но если вместо знаков «??????» оставить пустое место («заглушку»), то при L, равном 0, данный набросок будет работать — на выходе мы получим начальное разбиение, состоящее из единиц.

Пусть L равно 1. Обратимся к таблице 2.20. Проверяем *SmallSc*[8,1]. Если L больше или равно этому значению, то мы уменьшим значение L на *SmallSc*[8,1], увеличим значение j на единицу и, в конечном итоге, получим разбиение 2111111. Итак, что нам необходимо сделать? Да просто просмотреть строку массива *SmallSc* с номером *sc* и определить последний элемент строки, его номер в j, который меньше текущего значения L. Мы последовательно отсекаем разбиения числа *sc*, начинающиеся с чисел j. Ниже приведен полный текст процедуры.

```
Procedure GetWhByNum(L:LongInt);
 Var i,j,sc:Integer;
 Begin
  sc:=N;i:=1;
  While sc<>0 Do
   Begin
    j:=1;
    While L>=SmallSc[sc,j] Do
      Begin
       Dec(L,SmallSc[sc,j]);Inc(j);
      End;
    Now[i]:=j;sc:=sc-j;Inc(i);
```

```
 End;
 Now[0]:=i-1;{*Корректируем длину разбиения.*}
End;
```

Четвертая задача. По разбиению (*Now*) получить его номер.

Принцип прост. Брать текущее число из разбиения, начиная с первого, и смотреть на его «вклад» в количество номеров. Например, разбиение «22211». Определяем, сколько номеров приходится на первую 2, затем на вторую и т. д.

```
Function GetNumByWh:LongInt;
 Var i,jk,p:Integer;
     sc:LongInt;
 Begin
  sc:=1;{*Формируем номер разбиения.*}
  jk:=N;{*Номер строки в массиве SmallSc.*}
  For i:=1 To Now[0] Do
   Begin{*Просматриваем
         разбиение.*}
    For p:=0 To Now[i]-1 Do sc:=sc+SmallSc[jk,p];
       {*Значение числа из разбиения определяет
         число столбцов в массиве SmallSc.*}
    jk:=jk-Now[i];{*Изменяем номер строки.*}
   End;
  GetNumByWh:=sc;
End;
```

В таблице 2.21 приведена ручная трассировка функции *GetNumByWh* для *N*=8, *Now*=22211.

Таблица 2.21

$sc_{старое}$	$jk_{старое}$	*i*	*P*	$sc_{новое}$	$jk_{новое}$
1	8	1	0	1	
			1	2	6
2	6	2	0		
			1	3	4
3	4	3	0		
			1	4	2
4	2	4	0	4	1
4	1	5	0	4	0

Итак, номер определен, он равен 4.

2.2.5. Последовательности из нулей и единиц длины *N* без двух единиц подряд

Первая задача. Подсчет числа последовательностей из нулей и единиц длины N без двух единиц подряд.

В таблице 2.22 приведен пример для N=4. Таких последовательностей 8.

Таблица 2.22

0000	0010	0101	1001
0001	0100	1000	1010

Как подсчитать количество таких последовательностей (обозначим через $P(N)$)?

Пусть есть последовательность длины N. На первом месте в последовательностях записан ноль. Число этих последовательностей $P(N-1)$.

Рассмотрим случай, когда на первом месте записана единица. Тогда на втором месте обязательно должен быть записан 0. Число таких последовательностей $P(N-2)$. Итак, получили формулу:

$$P(N)=P(N-1)+P(N-2),$$

а это не что иное, как формула вычисления чисел Фибоначчи. Таким образом, число последовательностей длины N, в которых нет двух идущих подряд 1, равно N-му числу Фибоначчи. Ограничимся диапазоном чисел типа *LongInt* ($N \le 44$).

Для вычисления количества последовательностей при больших значениях N требуется использовать «длинную» арифметику, в частности процедуры сложения и вывода длинных чисел. А при наших ограничениях можно ввести результаты в массив констант.

```
Const NSmall=44;
ResSm: Array[0..NSmall] Of LongInt=(1, 2, 3, 5, 8,
    13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
    1597, 2584, 4181, 6765, 10946, 17711, 28657,
    46368, 75025, 121393, 196418, 317811, 514229,
    832040, 1346269, 2178309, 3524578, 5702887,
    9227465, 14930352, 24157817, 39088169,
    63245986, 102334155, 165580141, 267914296,
    433494437, 701408733, 1134903170, 1836311903);
```

Процедура вычисления количества последовательностей:

```
Function GetSmall(N: Integer): LongInt;
  Begin
```

```
If N>NSmall Then GetSmall:=MaxLongInt
           Else GetSmall:=ResSm[N];
End;
```

Вторая задача. Генерация последовательностей.

Рассмотрим пример, приведенный в таблице 2.23, для N=6 (номер последовательности, последовательность).

Таблица 2.23

0	000000	7	001010	14	100001
1	000001	8	010000	15	100010
2	000010	9	010001	16	100100
3	000100	10	010010	17	100101
4	000101	11	010100	18	101000
5	001000	12	010101	19	101001
6	001001	13	100000	20	101010

Принцип перехода к следующей последовательности.

Начинаем с конца последовательности, подпоследовательности типа ..0010... переходят в тип ...0100... и типа ..001010... в тип 010000....

Формализация этого принципа приведена в следующей процедуре. Для хранения последовательности используется массив *Now*.

```
Procedure GetNext;
 Var i: Integer;
  Begin
   i:=N;
   While (Now[i]=1) Or ((i>1) And (Now[i-1]=1)) Do
     Begin
      Now[i]:=0;Dec(i);
     End;
   Now[i]:=1;
  End;
```

Третья задача. По номеру получить последовательность.

Зададим себе простой вопрос. Если значение L больше, чем число последовательностей длины $N-1$, то что находится на первом месте (слева) в последовательности — 0 или 1? Ответ очевиден: 1. Вычтем из значения L число последовательностей длины $N-1$ и продолжим сравнение с новым значением L. В этом суть решения, которая реализована в процедуре GetWhByNum.

В таблице 2.24 приведена ручная трассировка процедуры GetWhByNum для N=6, L=15.

Таблица 2.24

L	*i*	*P*	*Now*(******)
15	1	13	1*****
2	2	8	10****
2	3	5	100***
2	4	3	1000**
2	5	2	10001*
0	6	1	100010

```
Procedure GetWhByNum(L: LongInt);
 Var  i: Integer;
 Begin
 For i:=1 To N Do
  If L>=GetSmall(N-i) Then
    Begin
     Now[i]:=1;
     L:=L-GetSmall(N-i);
    End
   Else Now[i]:=0;
 End;
```

Четвертая задача. По последовательности получить ее номер в соответствии с введенным отношением порядка.

Если в последовательности на месте i находится 1, то в номере есть вклад от числа всех последовательностей длины $N-i$. Итак, следует просмотреть последовательность и по значению 1 подсчитать сумму количества последовательностей соответствующих длин.

```
Function GetNumByWh: LongInt;
 Var   i: Integer;
       sc: LongInt;
 Begin
  sc:=1;
  For i:=1 To N Do If Now[i]=1 Then
   Inc (sc, GetSmall(N-i));
  GetNumByWh:=sc;
 End;
```

2.2.6. Подмножества

Проблема. Научиться работать с подмножествами N-элементного множества натуральных чисел от 1 до N такими, что каждое следующее подмножество отличалось бы от предыдущего только на один элемент.

Решаем традиционные четыре задачи:
подсчет количества подмножеств;
генерация подмножеств;
определение подмножества по номеру;
определение номера по подмножеству.

Первая задача. Подсчет количества подмножеств.

Количество подмножеств множества из N элементов равно $2\times N$. Эта задача проста, если значение N не очень большое. Однако уже при N=15 количество подмножеств 32768, поэтому тип данных *Integer* требуется использовать осторожно.

Пусть N меньше или равно 100. При таких значениях N необходимо использовать элементы «длинной» арифметики, а именно: умножение «длинного» числа на короткое (пятая задача 1-й главы, процедура *Mul*) и вывод «длинного» числа (вторая задача 1-й главы, процедура *SayTLong*).

Процедура вычисления количества подмножеств множества из N элементов.

Есть небольшой «нюанс». Возводить двойку в степень N — утомительное занятие, сократим его по возможности.

```
Procedure Calc;
 Var  i: Integer;
Tmp: TLong;
 Begin
  i:=N;{*Информация для размышления. Основанием
        системы счисления в нашей задаче является
        10000, а два в степени 13 равно 8192.*}
  Res[1]:=1; Res[0]:=1;{*Два в степени 0 равно 1.*}
  While i>0 Do
   Begin
    If i>13 Then
      Begin
       Mul(Res, 1 shl 13, Tmp); i:=i-13;
      End
     {*Напомним, что команда shl - сдвиг влево,итак
     1 сдвигается на 13 разрядов влево, т.е. мы
     умножаем на  2^13.А почему нельзя умножать на 2^14?*}
            Else
```

```
      Begin
       Mul(Res, 1 shl i, Tmp); i:=0;
      End;
    Res:=Tmp;
    End;
End;
```

Вторая задача. Генерация подмножеств.

Подмножества N-элементного множества натуральных чисел от 1 до N будем представлять двоичными последовательностями длины N — *Now*[1], *Now*[2], ..., *Now*[*N*] (массив *Now*).

Элемент *Now*[*i*], равный единице, говорит о том, что $N-i+1$ принадлежит подмножеству.

Таблица 2.25

Число	Двоичное представление числа (*B*)	Двоичная последовательность (*Now*)	Подмножество
0	0000	0000	[]
1	0001	0001	[1]
2	0010	0011	[1,2]
3	0011	0010	[2]
4	0100	0110	[2,3]
5	0101	0111	[1,2,3]
6	0110	0101	[1,3]
7	0111	0100	[3]
8	1000	1100	[3,4]
9	1001	1101	[1,3,4]
10	1010	1111	[1,2,3,4]
11	1011	1110	[2,3,4]
12	1100	1010	[2,4]
13	1101	1011	[1,2,4]
14	1110	1001	[1,4]
15	1111	1000	[4]

Во втором столбце таблицы 2.25 записаны двоичные представления чисел от 0 до 2^4-1. Каждое из чисел мы подвергли преобразованию (кодирование по Грею), заменив каждую двоичную цифру, кроме первой, на ее сумму с предыдущей цифрой по модулю 2, таким образом, получив элементы массива *Now*. Т. е., *Now*[1]:=*B*[1] и *Now*[*i*]:=*B*[*i*]⊕*B*[*i*−1] для всех i от 2 до N.

В четвертом столбце таблицы 2.25 записаны подмножества, построенные по единичным элементам массива *Now*. Обратите внимание на то, что каждое следующее отличается от предыдущего только на один элемент.

Запишем формирование массива *Now* в виде процедуры.

```
Procedure TransferB;
 Var i: Integer;
 Begin
  FillChar(Now, SizeOf(Now), 0);
  Now[1]:=B[1];
  For i:=2 To N Do Now[i]:=B[i] Xor B[i-1];
 End;
```

Обратное преобразование:
B[1]:=Now[1], B[i]:=Now[i]⊕B[i-1] для i от 2 до N.

```
Procedure TransferToB;
 Var i, j: Integer;
 Begin
  FillChar(B, SizeOf(B), 0);
  B[1]:=Now[1];
  For i:=2 To N Do B[i]:=Now[i] Xor B[i-1];
 End;
```

Воспринимая данное преобразование как факт, осмысление которого лежит за пределами данного материала, генерацию подмножеств можно осуществлять с использованием массива *B*.

В качестве материала для самостоятельного осмысления предлагается следующая изящная процедура генерации очередного разбиения.

```
Procedure GetNext;
  Var i: Integer;
   Begin
     i:=N+1;
      Repeat
        Dec(i);
        B[i]:=B[i] Xor 1;
      Until B[i]=1;
     Now[i]:=Now[i] Xor 1;
   End;{*Будет ли работать данная процедура для
         последовательности 1111 (N=4)?*}
```

Можно решать задачу по-другому. Вспомним зрительный образ «лесенки» со стрелками, который мы использовали при

генерации перестановок, таких, что каждая следующая отличалась от предыдущей только одной транспозицией соседних элементов. В данной задаче следует использовать не лесенку, а прямоугольное поле $2\times N$. Шашки в начальный момент находятся в первой строке (она соответствует нулевым значениям в двоичной последовательности) и двигаются в направлении стрелок. Определяется первая справа шашка, которую можно сдвинуть в направлении стрелки. Сдвигаем ее, а у шашек, находящихся справа от нее, меняем направление движения. Последовательности 0000 при $N=4$ соответствует следующая «картинка» — рисунок 2.5.

				1
↑	↑	↑	↑	0

Рис. 2.5

А последовательности 0100 соответствует рисунок 2.6.

	↑			1
↑		↓	↓	0

Рис. 2.6

Следующее состояние поля будет иметь вид (последовательность 1100), приведенный на рисунке 2.7.

↑	↓			1
		↑	↑	0

Рис. 2.7

Процесс генерации окончится при состоянии (последовательность 1000), изображенном на рисунке 2.8.

↑				1
	↓	↓	↓	0

Рис. 2.8

Третья задача. Определение подмножества по номеру.

Получение последовательности B по номеру разбиения на подмножества L соответствует записи числа L в двоичной системе счисления с использованием массива B. Двоичный разряд числа L — это значение соответствующего элемента массива B.

```
Procedure GetWhByNum(L: LongInt);
 Var i: Integer;
 Begin
  For i:=1 To N Do
    Begin
      B[N-i+1]:=L And 1;{*Выделяем младший разряд
            числа L.*}
      L:=L shr 1;{*Уменьшаем L в два раза (деление
            на 2).*}
    End;
 End;
```

Четвертая задача. Определение номера по подмножеству.

Алгоритм получения номера разбиения на подмножества по последовательности B соответствует в нашем случае переводу числа из двоичной системы счисления в десятичную.

```
Function GetNumByWh: LongInt;
 Var  sc: LongInt;
      i: Integer;
 Begin
  sc:=0;
  For i:=1 To N Do sc:=sc*2 + B[i];
  GetNumByWh:=sc+1;
 End;
```

2.2.7. Скобочные последовательности

Последовательность из $2{\times}N$ скобок правильная, если для любого значения i ($1 \le i \le 2{\times}N$), число открывающих скобок, например «(», больше или равно числу закрывающих скобок «)» и общее число открывающих скобок в последовательности равно числу закрывающих.

Пример. Последовательность (())()() — правильная (N=4), последовательность ((()))(() — не является правильной.

Мы по-прежнему решаем наши *четыре задачи*. Однако изменим порядок их рассмотрения. Начнем с генерации последовательностей.

Пусть N равно 4. В нечетных столбцах таблицы 2.26 приведены номера последовательностей, а в соседних — соответствующие последовательности.

Таблица 2.26

0	()()()()	7	(()())()
1	()()(())	8	(()(()))
2	()(())()	9	(()(()))
3	()(()())	10	((()))()
4	()((()))	11	((())())
5	(())()()	12	((()()))
6	(())(())	13	((())())

Что же мы делаем? Какое отношение порядка введено на множестве последовательностей? Сколько существует таких последовательностей для заданного значения N? Вопросов много, ответов пока нет.

Пусть наша последовательность хранится в переменной строкового типа *Now*. Мы просматриваем строку справа налево до первой комбинации скобок типа «)(». Она обязательно должна встретиться, ибо, считаем, что текущая последовательность не является последней. Заменяем эту комбинацию скобок на «()». Подсчитываем с начала строки (последовательности), на сколько число открывающих скобок больше числа закрывающих, дописываем это количество закрывающих скобок, и если строка не длины $2\times N$, то дополняем ее комбинациями скобок «()».

```
Procedure GetNext;
 Var i, j, sc: Integer;
 Begin
  i:=N*2;
  While (i>1) And (Now[i-1] + Now[i]<>')(')
      Do Dec(i);{*Поиск комбинации ")(".*}
  Now[i-1]:='('; Now[i]:=')';{*Замена.*}
      Now:=Copy(Now,1,i);
  sc:=0;{*Подсчет разности числа открывающих и
      числа закрывающих скобок.*}
  For j:=1 To i Do
   If Now[j]='('
     Then Inc(sc)
     Else Dec(sc);
  While sc<>0 Do
   Begin
```

```
    {*Дополнение подпоследовательности
    закрывающими скобками.*}
    Now:=Now+')';Dec(sc);
   End;
While Length(Now)<2*N Do Now:=Now+'()';
    {*Дополняем строку, максимально "ухудшая"
     ее, т.е. из остатка строки делаем ее
     начальное значение.*}
End;
```

Даны две последовательности P_1 и P_2, например ()()(()) и ()(()()). Какая из них имеет больший номер? Просматриваем последовательности слева направо, сравнивая соответствующие символы. Найдем первое значение i, при котором $P_1[i]<>P_2[i]$. Если окажется, что $P_1[i]=')'$, а $P_2[i]='('$, то P_1 имеет меньший номер, чем P_2. Можно записать сей факт и в математических обозначениях, но суть не изменится, поэтому воздержимся.

Перейдем к *первой задаче*. Введем понятие частично правильной скобочной последовательности. Последовательность частично правильная, если для любой позиции в последовательности число открывающих скобок больше или равно числу закрывающих, естественно, что количество тех и других считается от начала последовательности. Так, последовательность «(((((» является частично правильной, а последовательность «(()))((» — нет, для позиции 5 число закрывающих больше числа открывающих скобок.

Оформим наши дальнейшие рассуждения в виде таблицы 2.7. Номер строки указывает на число скобок в последовательности, номер столбца — на разность между числом открывающих и закрывающих скобок, имя таблицы *ScSmall*, элемент таблицы *ScSmall*[*i*,*j*] равен количеству частично правильных последовательностей из i скобок, причем разность между числом открывающих и закрывающих равна j. Это ключевой момент для понимания: «разность — номер столбца». На диагонали таблицы 2.7 записаны 1, число последовательностей, состоящих из одних открывающих скобок, а они попадают под определение частично правильной последовательности. Элементы таблицы 2.7 *ScSmall*[*i*,*j*] равны 0, если i и j числа разной четности.

Примеры.
ScSmall[3,1]=2: (((), ()(.
ScSmall[4,2]=3: ((((), (()(, ()((.
ScSmall[4,0]=2: ()(), (()).
ScSmall[5,1]=5: ()()(, (())(, ((())), (()(), ()(().

И «крохотный факт»:

ScSmall[5,1]=*ScSmall*[4,0]+*ScSmall*[4,2].

Дописываем в конец последовательностей из *ScSmall*[4,0] открывающую скобку, а в конец *ScSmall*[4,2] — закрывающую. А это уже реальный путь подсчета числа последовательностей. Ответом задачи будет *ScSmall*[2×N,0].

Таблица 2.27

	–1	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	0	1	0	1	0	0	0	0	0	0
3	0	0	2	0	1	0	0	0	0	0
4	0	2	0	3	0	1	0	0	0	0
5	0	0	5	0	4	0	1	0	0	0
6	0	5	0	9	0	5	0	1	0	0
7	0	0	14	0	14	0	6	0	1	0
8	0	14	0	28	0	20	0	7	0	1

Ограничимся при вычислениях диапазоном типа *LongInt*.

```
Const SmallN=37;
Var ScSmall: Array[-1..SmallN + 1, -1..SmallN + 1]
    Of LongInt;
```

Процедура формирования таблицы вряд ли нуждается в пояснениях.

```
Procedure FillSmallSc;
 Var i, j: Integer;
 Begin
  FillChar(ScSmall, SizeOf(ScSmall), 0);
  ScSmall[0, 0]:=1;
  For i:=1 To SmallN-1 Do
   For j:=0 To SmallN Do
     ScSmall[i,j]:=ScSmall[i-1,j-1]+
     ScSmall[i-1,j+1];
 End;
```

Дополним ее функцией определения количества частично правильных последовательностей из массива *SCSmall*.

```
 Function GetSmallSc(N, Up: LongInt): LongInt;
  Begin
   If (N<0) Or (Up<0)
```

```
    Then GetSmallSc:=0
    Else
   If (N>SmallN) Or (Up>SmallN)
    Then GetSmallSc :=MaxLongInt
    Else GetSmallSc:=ScSmall[N, Up];
End;
```

Вычисление последовательности по номеру и обратное преобразование можно вынести на самостоятельную часть работы. Ниже приведены соответствующие процедура и функция. В переменной *Up*, как и выше, формируется текущая разность между числом открывающих и закрывающих скобок.

```
Procedure GetWhByNum(L: LongInt);
 Var  i, Up: Integer;
      P: LongInt;
 Begin
  Now:=''; Up:=0;
  For i:=1 To N*2 Do
    Begin
      P:=GetSmallSc(N*2-i, Up-1);
      If (L>=P) Then
        Begin
          Now:=Now+'('; Inc(Up); L:=L-P;
        End
                Else
        Begin
          Dec(Up); Now:=Now+')';
        End;
    End;
 End;

Function GetNumByWh: LongInt;
 Var     sc: LongInt;
        i, Up: Integer;
 Begin
   sc:=1; Up:=0;
   For i:=1 To N*2 Do
     If Now[i]='(' Then
       Begin
         sc:=sc+GetSmallSc(N*2-i,Up-1);Inc(Up);
       End
```

```
    Else Dec(Up);
   GetNumByWh:=sc;
End;
```

2.3. Задачи

1. Разработать программу генерации всех последовательностей длины k из чисел 1, 2, ... N. Первой последовательностью является 1, 1, ...,1, последней — N, N, ..., N.

2. Разработать программу генерации всех последовательностей длины k, у которых i-й элемент не превосходит значения i. Первой последовательностью является 1, 1, ...,1, последней — 1, 2, ..., k.

3. Перечислить все разбиения натурального числа N на натуральные слагаемые (разбиения, отличающиеся лишь порядком слагаемых, считаются за одно) в следующих порядках например, при N=4:

4, 3+1, 2+2, 2+1+1, 1+1+1+1;
4, 2+2, 1+3, 1+1+2, 1+1+1+1;
1+1+1+1, 1+1+2, 1+3, 2+2, 4.

4. Разработать программу генерации всех последовательностей длины $2\times N$, составленных из N единиц и N минус единиц, у которых сумма любого начального отрезка неотрицательна, т. е. количество минус единиц в нем не превосходит количества единиц.

5. Определить, что делает следующая процедура. Первое обращение — *R(N,1)*.

```
Procedure R(t,k:Integer);
 Var i:Integer;
 Begin
  If t=1 Then
   Begin
    A[k]:=1;<вывод k элементов массива A>;
   End
         Else
   Begin
     A[k]:=t;<вывод k элементов массива A>;
     For i:=1 To t-1 Do
      Begin A[k]:=t-i;R(i,k+1);End;
   End;
End;
```

6. На множестве перестановок задается антилексикографический порядок. Решить задачи 2–5 из п. 2.2.1.

7. Решить задачи 1–5 из п. 2.2.1 для перестановок с повторениями.

8. Решить задачи 1–5 из п. 2.2.2 для размещений с повторениями.

9. Решить задачи 1–5 из п. 2.2.3 для сочетаний с повторениями.

10. Путем трассировки приведенной программы [18] определите ее назначение и алгоритм работы.

```
Const n=4;
Var A:Array[1..n] Of Integer;
    i:Integer;
Function Place(i,m:Integer):Integer;
 Begin
  If (m Mod 2=0) And (m>2)
   Then
    If i<m-1
     Then Place:=i
     Else Place:=m-2
   Else Place:=m-1;
 End;
Procedure Perm(m:Integer);
 Var i,t:Integer;
 Begin
  If m=1 Then
   Begin
     For i:=1 To n Do Write(A[i]:3); WriteLn;
   End
         Else
     For i:=1 To m Do
      Begin
       Perm(m-1);
       If i<m Then
         Begin
          t:=A[Place(i,m)]; A[Place(i,m)]:=A[m];A[m]:=t;
         End;
      End;
 End;
Begin
 For i:=1 To n Do A[i]:=i;
```

```
 Perm(n);
End.
```

11. Путем трассировки приведенной программы [18] определите ее назначение и алгоритм работы.

```
Const n=4;
Var A:Array[1..n] Of Integer;
   i,j,p,l:Integer;
Begin
 For i:=1 To n Do A[i]:=0;
 i:=0;
 Repeat
  For l:=1 To n Do Write(A[l]:3);
  WriteLn;
  i:=i+1;p:=1;j:=i;
  While j Mod 2=0 Do
   Begin
      j:=j Div 2;p:=p+1;
   End;
  If p<=n Then A[p]:=1-A[p];
 Until p>n;
End.
```

12. Путем трассировки приведенной программы [18] определите ее назначение и алгоритм работы.

```
Const n=5;
      k=3;
 Var A:Array[1..n] Of Integer;
    i,j,p:Integer;
 Begin
  For i:=1 To k Do A[i]:=i;
  p:=k;
  While p>=1 Do
   Begin
     For i:=1 To k Do Write(A[i]:3);
     WriteLn;
     If A[k]=n Then p:=p-1 Else p:=k;
     If p>=1 Then
     For i:=k DownTo p Do A[i]:=A[p]+i-p+1;
  End;
 End.
```

13. «*Взрывоопасность*». На одном из секретных заводов осуществляется обработка радиоактивных материалов, в результате которой образуются радиоактивные отходы двух типов: *А* (особо опасные) и *Б* (неопасные). Все отходы упаковываются в специальные прямоугольные контейнеры одинаковых размеров, после чего эти контейнеры укладываются в стопку один над другим для сохранения. Стопка является взрывоопасной, если в ней соседствуют два ящика с отходами типа *А*.

Требуется написать программу, которая подсчитывает количество возможных вариантов формирования невзрывоопасной стопки из заданного общего числа контейнеров N ($1 \le N \le 100$).

Входные данные. Во входном файле *input.txt* содержится единственное число N.

Выходные данные. В выходной файл *output.txt* необходимо вывести искомое число вариантов.

Пример входных и выходных данных представлен на рисунке 2.9.

Пример.
input.txt 4
output.txt 8

Рис. 2.9

Указание. Обозначим опасный контейнер 1, а неопасный — 0. Моделью стопки является последовательность из 0 и 1. Взрывоопасная стопка есть последовательность из 0 и 1, в которой соседствуют хотя бы две 1 подряд. Задача сводится к подсчету числа последовательностей длины N без двух соседних 1. Приведем пример. При N=4 таких последовательностей 8, они представлены в таблице 2.28.

Таблица 2.28

0000	0010	0101	1001
0001	0100	1000	1010

Как подсчитать количество таких последовательностей (обозначим через $P(N)$)?

Пусть есть последовательность длины N. На первом месте в последовательности записан 0. Число таких последовательностей $P(N-1)$.

Рассмотрим случай, когда на первом месте записана 1. Тогда на втором месте обязательно должен быть записан 0. Число таких последовательностей $P(N-2)$.

Итак, получили формулу: $P(N)=P(N-1)+P(N-2)$, а это не что иное, как формула вычисления чисел Фибоначчи. Таким образом, число последовательностей длины N, в которых нет двух

идущих подряд 1, равно N-му числу Фибоначчи. Тип данных *LongInt* достаточен только для вычислений при $N \le 44$. Для вычисления количества последовательностей при бóльших значениях N требуется использовать «длинную» арифметику, а именно процедуры сложения и вывода «длинных» чисел.

Если задачу чуть-чуть изменить — вычислять количество взрывоопасных стопок, то потребуется процедура вычитания «длинных» чисел. Из 2^N необходимо вычесть число невзрывоопасных стопок.

14*. Натуральные числа от 1 до N ($1 \le N \le 13$) поступают на обработку в определенной последовательности. Например, 3, 4, 6, 2, 5, 1. Числа размещаются в таблице следующим образом: последовательно просматриваются числа из первой строки и сравниваются с поступившим числом k. Если число k больше всех чисел строки, то оно размещается в конце строки. Если же найдено (первое) число t, такое, что $t>k$, то число k записывается на это место, а число t «выдавливается» из строки и размещается во второй строке по этому же принципу, затем в третьей (если потребуется) и т .д., пока для числа (k или «выдавливаемого») не будет найдено место. Для приведенной последовательности вид размещения представлен на рисунке 2.10.

```
1 4 5
2 6
3
```

Рис. 2.10

Требуется по конечному результату восстановить все возможные входные последовательности чисел, обработка которых по данному алгоритму приводит к этому размещению. Так, для примера из формулировки задачи, ответом являются 16 последовательностей, приведенных на рисунке 2.11.

```
3 2 1 4 6 5
3 2 1 6 4 5
3 4 2 1 6 5
3 2 4 1 6 5
3 2 6 1 4 5
3 6 2 1 4 5
3 4 2 6 1 5
3 4 6 2 1 5
3 2 4 6 1 5
3 2 6 4 1 5
3 6 2 4 1 5
3 4 2 6 5 1
3 4 6 2 5 1
3 2 4 6 5 1
3 2 6 4 5 1
3 6 2 4 5 1
```

Рис. 2.11

Указание. Простейший способ решения заключается в генерации всех перестановок и проверке — удовлетворяет ли очередная перестановка «правилам игры». Однако если ввести ограничение на время решения, то этот способ не выдерживает элементарной критики. Обратим внимание на следующие моменты. На обработку поступила последовательность (N=6) 1, 2, 3, 4, 5, 6. Ее размещение имеет вид: 1 2 3 4 5 6. Обратный порядок по-

---

* Задача с международной олимпиады школьников 2001 г. (формулировка сокращена).

ступления дает размещение по одному элементу в строке: в первой — 1, во второй — 2 и т. д. Зададим себе вопрос: а возможны ли размещения, при которых элементы в столбцах (не в строках!) идут не в порядке возрастания, как, например, приведенное на рисунке 2.12. Оказывается, нет.** Мы всегда получаем размещения, элементы в столбцах и строках которых записаны в порядке возрастания. Если так, то первое решение получается простым выписыванием элементов по столбцам в обратном порядке. Для примера из формулировки задачи это будет 3 2 1 6 4 5.

1	4	7	12
2	6	8	
3	5	9	
10	11		

Рис. 2.12

Второй вариант решения может заключаться в том, что из полученной (первой) последовательности путем перестановки соседних элементов мы получаем какую-то последовательность. Проверяем ее на «пригодность» и если она удовлетворяет «условиям игры», то запоминаем ее в очереди. Итак, выполняем все перестановки соседних элементов в исходной последовательности, а затем переходим к очередному необработанному элементу очереди, если он есть.

Недостаток алгоритма очевиден. Мы можем получать одну и ту же последовательность несколькими способами. Поэтому приходится их хранить и перед записью в очередь проверять, нет ли дублирования. А это уже потеря времени и, естественно, данное решение (приведем его «костяк») является только очередным приближением, хотя и работающим, к окончательному варианту.

```
Const NMax=13;
 Type MyArray=Array[1..NMax] Of Word;
 Var A:MyArray;
  Function Check(Const P:MyArray):Boolean;
    {*Функция проверки – удовлетворяет ли очередная
      последовательность условиям задачи?*}
   Begin
...
   End;

 Procedure Solve;
  Type pnt=^Ilem;
```

** В этом заключается неточность в формулировки задачи на международной олимпиаде. Если проверять по несуществующим размещениям, то часть решений, включая опубликованные в газете «Информатика», будут генерировать неизвестные входные последовательности.

```
    Ilem=Record{*Описываем элемент очереди.
      Очередь размещаем "в куче". Обычный список.
      Записываем в конец - указатель ykw,
      считываем первый элемент - указатель ykr.*}
        Next:pnt;
        Data:MyArray
       End;
Var R,Q:MyArray;
      ykr,ykw, p:pnt;
      i:Word;
      head:pnt;
Function NotQue:Boolean;{*Функция проверки -
  есть ли данная последовательность в очереди?*}
 Begin
  ...
 End;
Begin
 New(ykw);ykw^.Data:=A;ykw^.Next:=Nil;head:=ykw;
   {*Записываем первый элемент в очередь.*}
 ykr:=head;
 While ykr<>nil Do
   Begin {*Пока очередь не пуста.*}
    R:=ykr^.Data;{*Берем элемент из очереди.*}
    For i:=1 To N-1 Do
     Begin
     {*Генерируем все возможные последовательности
      путем перестановок соседних элементов.*}
      Q:=R;
      Swap(Q[i],Q[i+1]);
      If Check(Q) And NotQue Then
       Begin
       {*Если последовательность удовлетворяет
       условиям задачи и ее нет в очереди, то
       записываем в очередь.*}
        New(p);p^.Data:=Q;p^.Next:=Nil;
        ykw^.Next:=p;
        ykw:=p;
        Print(Q);{*Процедура вывода элементов
        одномерного массива в файл не приводится.*}
       End;
     End;
     ykr:=ykr^.next;{*Переходим к следующему
                     элементу очереди.*}
   End;
```

```
 End;
Begin
{*Основная программа. Ввод данных. Формирование
первой последовательности массив A). *}

  Solve;

End.
```

15. На клетчатой полоске бумаги высотой в одну клеточку и длиной N клеток некоторые клетки раскрашены в зеленый и белый цвета. По этой полоске строится ее код, которым является последовательность чисел — количества подряд идущих зеленых клеток слева направо. Например, для полоски на рисунке 2.13

Рис. 2.13

кодом будет последовательность 2 3 2 8 1. При этом количество белых клеток, которыми разделяются группы зеленых клеток, нигде не учитывается (главное, что две группы разделяются, по крайней мере, одной белой клеткой). Одному и тому же коду могут соответствовать несколько полосок. Например, указанному коду соответствует полоса, изображенная на рисунке 2.14:

Рис. 2.14

Задача состоит в том, чтобы найти количество полосок длины N, соответствующих заданному коду.

Входные данные. В единственной строке входного файла *input.txt* записано число N — длина полоски ($1 \le N \le 200$), затем число K — количество чисел в коде ($0 \le K \le (N+1)/2$) и далее K чисел, определяющих код.

Выходные данные. В выходной файл *output.txt* записывается количество полосок длины N, соответствующих заданному коду.

Указание. Добавляем к каждому числу кода по единице (соответствует пробелу), получаем

Примеры
input.txt 4 0
output.txt 1
input.txt 5 2 1 2
output.txt 3
input.txt 4 2 2 2
output.txt 0

Рис. 2.15

значение s. Остается решить задачу о подсчете числа размещений $N-s$ пробелов по известному количеству мест — количеству чисел в коде плюс единица. Естественно, что ограничения требуют применения многоразрядной арифметики.

На рисунке 2.15 приведены возможные значения входных данных (файл *input.txt*) и соответствубщие значения выходных данных (*output.txt*).

16. Между числами $a_1,\ldots,a_N$ нужно вставить знаки арифметических операций +, –, * так, чтобы результат вычисления полученного арифметического выражения был равен заданному целому числу B.

Входные данные. Во входном файле *input.txt* записано сначала требуемое число B ($-2\times10^9 \le B \le 2\times10^9$), затем число N — количество чисел в арифметическом выражении ($1 \le N \le 12$), а затем целые числа $a_1,\ldots,a_N$ ($0 \le a_i \le 20$)

Выходные данные. В выходной файл *output.txt* вывести искомое арифметическое выражение или число 0, если его построить не удается. Если решений несколько, вывести любое из них.

Указание. Количество мест для расстановки знаков — $N-1$. Требуется сгенерировать все строки этой длины, состоящие из символов +, –, *, их 3^{N-1}. Для каждой строки вычислить арифметическое выражение и, в зависимости от результата, продолжить процесс генерации или закончить работу.

Примеры

input.txt
1053
6
5 6 7 10 15 8

output.txt
5+6+7*10*15–8

input.txt
12
2
1 2

output.txt
0

Рис. 2.16

На рисунке 2.16 приведены возможные значения входных данных (файл *input*) и соответствующие значения выходных данных (файл *output*).

Представляет определенный интерес алгоритм вычисления выражения. Приведем его.

В строке B (глобальная переменная относительно приведенных функций) записана очередная последовательность из символов +, –, *, а в массиве A — последовательность чисел.

```
Function GetRes: Comp;
  Var tm: Comp;
      p:Integer;
```

```
Begin
  p:=1;
  tm:=GetMul(p);
  While (p<N) And (B[p] In ['+', '-']) Do
   Begin
    Inc(p);
    Case B[p-1] Of
      '+': tm:=tm+GetMul(p);
      '-': tm:=tm-GetMul(p);
    End;
   End;
  GetRes:=tm;
End;
```

И функция, вычисляющая произведение очередных сомножителей.

```
Function GetMul(Var p: Integer): Comp;
 Var tm: Comp;
  Begin
   tm:=A[p];
    While (p<N) And (B[p]='*') Do
      Begin
        tm:=tm*A[p+1];
        Inc(p);
      End;
    GetMul:=tm;
  End;
```

Обратной задачей является поиск количества «счастливых билетов» (например, троллейбусных, т. е. шестизначных). Общая формулировка задачи. Последовательность из $2\times N$ цифр, каждая цифра от 0 до 9, называется счастливым билетом, если сумма первых N цифр равна сумме последних N цифр. Операции даны, требуется найти числа.

Пусть нам необходимо подсчитать количество счастливых чисел в диапазоне *[A,B]*, где A, B — заданные натуральные числа, имеющие в десятичной записи не более 20 цифр. Пусть N=3, A=1, B=999999. Ответ задачи — 55251.

Следующая функция является наброском решения. Она правильная, но работает достаточно медленно.

```
Function Rec(k,Sum:LongInt):Comp;{*Параметрами
  рекурсии являются k - номер позиции в числе и
  Sum - сумма цифр в числе до позиции k.*}
```

```
Var i:Integer;
    Ans:LongInt;
Begin
 If k=2*N+1
   Then
    If Sum=0 Then Rec:=1 Else Rec:=0
    Else
     Begin
      Ans:=0;
      For i:=0 To 9 Do
       If k<=N
        Then Ans:=Ans+Rec(k+1,Sum+i)
        Else
         If i<=Sum Then Ans:=Ans+Rec(k+1,Sum-i);
       Rec:=Ans;
     End;
End;
```

Работа алгоритма ускоряется за счет следующего приема. Схема рекурсивного подсчета замедляется из-за многократного подсчета одного и того же значения, например *Rec*(3,8) при N=2. Следует ввести структуру данных для хранения подсчитанных значений сумм типа

```
M:Array[1..20,0..90] Of Comp.
```

Пусть начальное значение элементов массива равно −1. После того, как вычислено какое-то значение, оно запоминается в массиве и используется при дальнейшей работе. Проведите соответствующее изменение алгоритма.

17. На плоскости заданы N точек с целочисленными координатами. Требуется выделить те точки этого множества, которые образуют выпуклую оболочку (вырожденные случаи — все точки на одной прямой и т. д. не рассматриваются).

Указание. Специальные методы построения выпуклой оболочки будут рассмотрены в главе 5. В данном случае можно воспользоваться следующим фактом. Точка (x, y) принадлежит выпуклой оболочке, если она не лежит внутри любого треугольника, образованного остальными точками исходного множества. Всего треугольников, которые можно построить из N точек, — C_N^3, общая временная сложность задачи $O(N^4)$.

3. Перебор и методы его сокращения

Обучение программированию, алгоритмизации — сверхсложное дело*. Знание конструкций языка программирования, способов описания алгоритмов не решает проблемы. Требуется проделать огромную работу для того, чтобы сформировать стиль мышления, присущий специалисту по информатике. Этот стиль характеризуется, на наш взгляд, двумя основополагающими моментами: язык информатики (он не зависит от конкретного языка программирования) должен стать естественным языком выражения своих мыслей (рассуждений); методология решения задач, присущая информатике, должна быть освоена не на фактографическом уровне. Тема «перебор и методы его сокращения» является одной из ключевых в этой системе обучения. Весь процесс обучения построен через решение задач, естественно, с использованием компьютера.

3.1. Перебор с возвратом (общая схема)

Даны N уопрядоченных множеств U_1, U_2, ..., U_N (N — известно), и требуется построить вектор $A=(a_1, a_2, ..., a_N)$, где $a_1 \in U_1$, $a_2 \in U_2$, ..., $a_N \in U_N$, удовлетворяющий заданному множеству условий и ограничений.

В алгоритме перебора вектор A строится покомпонентно слева направо. Предположим, что уже найдены значения первых $k-1$ компонент, $A=(a_1, a_2, ..., a_{k-1}, ?, ..., ?)$, тогда заданное множество условий ограничивает выбор следующей компоненты a_k некоторым множеством $S_k \subset U_k$. Если $S_k <> [\ ]$ (не пустое), мы вправе выбрать в качестве a_k наименьший элемент S_k и перейти к выбору $k+1$ компоненты и так далее. Однако если условия таковы, что S_k оказалось пустым, то мы возвращаемся к выбору $k-1$ компоненты, отбрасываем a_{k-1} и выбираем в качестве нового a_{k-1} тот элемент S_{k-1}, который непосредственно следует за только что отброшенным. Может оказаться, что для нового a_{k-1} условия задачи допускают непустое S_k, и тогда мы пытаемся

---

* Может быть, это явилось одной из причин, по которой, после появления мощных персональных компьютеров, образовательная информатика с таким энтузиазмом «бросилась» в изучение так называемых информационных технологий.

снова выбрать элемент a_k. Если невозможно выбрать a_{k-1}, мы возвращаемся еще на шаг назад и выбираем новый элемент a_{k-2} и так далее.

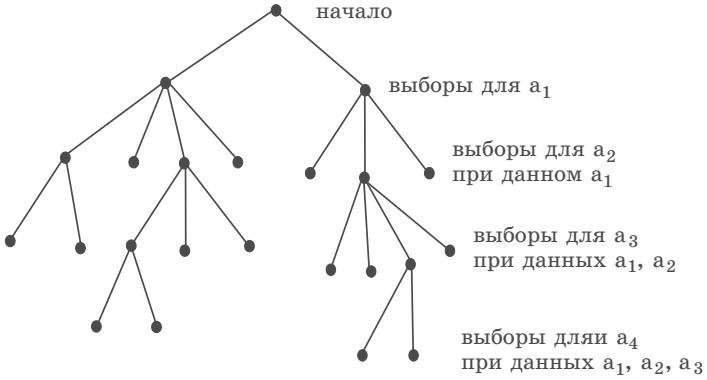


Рис. 3.1

Графическое изображение — дерево поиска (см. рисунок 3.1). Корень дерева (0 уровень) есть пустой вектор. Его сыновья суть множество кандидатов для выбора a_1, и, в общем случае, узлы k-го уровня являются кандидатами на выбор a_k при условии, что a_1, a_2, ..., a_{k-1} выбраны так, как указывают предки этих узлов. Вопрос о том, имеет ли задача решение, равносилен вопросу, являются ли какие-нибудь узлы дерева решениями. Разыскивая все решения, мы хотим получить все такие узлы.

Рекурсивная схема реализации алгоритма.

```
Procedure Backtrack(<вектор,i>);
 Begin
  If <вектор является решением>
   Then <записать его>
   Else
    Begin
      <вычислить Si>;
       For <a∈Si>Do Backtrack(<вектор||a>,i+1);
            {*|| - добавление к вектору компоненты.*}
    End;
 End;
```

Оценим временную сложность алгоритма. Данная схема реализации перебора приводит к экспоненциальным алгоритмам. Действительно, пусть все решения имеют длину N, тогда иссле-

довать требуется порядка $|U_1| \times |U_2| \times ... \times |U_N|$ узлов дерева. Если значение U_i ограничено некоторой константой C, то получаем порядка C^N узлов.

3.2. Примеры задач для разбора общей схемы перебора

1. «Задача о расстановке ферзей».

Для разбора общей схемы одной из лучших задач является задача о расстановке ферзей. На шахматной доске $N \times N$ требуется расставить N ферзей таким образом, чтобы ни один ферзь не атаковал другого.

Отметим следующее.

Все возможные способы расстановки ферзей — $C_{N^2}^N$ (для N=8 около $4{,}4 \times 10^9$).

Каждый столбец содержит самое большее одного ферзя, что дает только N^N расстановок (для N=8 $1{,}7 \times 10^7$).

Никакие два ферзя нельзя поставить в одну строку, а поэтому, для того чтобы вектор (a_1, a_2, ..., a_N) был решением, он должен быть перестановкой элементов (1, 2, ..., N), что дает только $N!$ (для N=8 $4{,}0 \times 10^4$) возможностей.

Никакие два ферзя не могут находиться на одной диагонали, это сокращает число возможностей еще больше (для N=8 в дереве остается 2056 узлов).

Итак, с помощью ряда наблюдений мы исключили из рассмотрения большое число возможных расстановок N ферзей на доске размером $N \times N$. Использование подобного анализа для сокращения процесса перебора называется поиском с ограничениями или отсечением ветвей в связи с тем, что при этом удаляются поддеревья из дерева.

Второе. Другим усовершенствованием является слияние, или склеивание, ветвей. Идея состоит в том, чтобы избежать выполнения дважды одной и той же работы: если два или больше поддеревьев данного дерева изоморфны, мы хотим исследовать только одно из них. В задаче о ферзях мы можем использовать склеивание, заметив, что если $a_1 > \lceil \frac{N}{2} \rceil$, то найденное решение можно отразить и получить решение, для которого $a_1 \le \lceil \frac{N}{2} \rceil$. Следовательно, деревья, соответствующие, например, случаям a_1=2 и a_1=N−1, *изоморфны*.

Рисунок 3.2 иллюстрирует сказанное и поясняет ввод используемых структур данных.

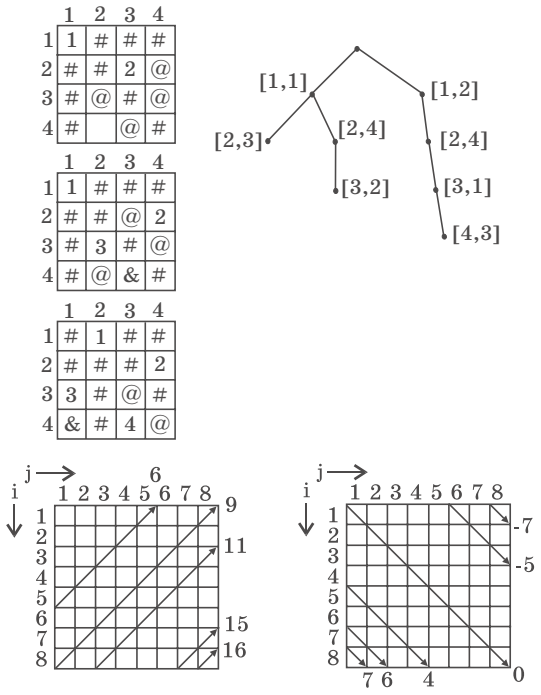


Рис. 3.2

Структуры данных.

```
Up:Array[2..16] Of Boolean;{*Признак занятости
                диагоналей первого типа.*}
Down:Array[-7..7] Of Boolean;{*Признак занятости
                диагоналей второго типа.*}
Vr:Array[1..8] Of Boolean;{*Признак занятости
                вертикали.*}
X:Array[1..8] Of Integer;{*Номер вертикали,
    на которой стоит ферзь на каждой горизонтали.*}
```

Далее идет объяснение «кирпичиков», из которых «складывается» решение (технология «снизу вверх»).

```
Procedure Hod(i,j:Integer); {*Сделать ход.*}
 Begin
   X[i]:=j;Vr[j]:=False;Up[i+j]:=False;
   Down[i-j]:=False;
 End;
Procedure O_hod(i,j:Integer);{*Отменить ход.*}
```

```
 Begin
   Vr[j]:=True;Up[i+j]:=True;Down[i-j]:=True;
  End;
Function D_hod(i,j:Integer):Boolean;
{*Проверка допустимости хода в позицию (i,j).*}
 Begin
   D_hod:=Vr[j] And Up[i+j] And Down[i-j];
 End;
```

Основная процедура поиска одного варианта расстановки ферзей имеет вид:

```
Procedure Solve(i:Integer;Var q:Boolean);
 Var j:Integer;
 Begin
  j:=0;
  Repeat
   Inc(j);q:=False;{*Цикл по вертикали.*}
   If D_hod(i,j) Then
    Begin
     Hod(i,j);
     If i<8
      Then
       Begin
        Solve(i+1,q);
        If Not q
         Then O_hod(i,j);
       End
      Else q:=True;{*Решение найдено.*}
    End;
  Until q Or (j=8);
 End;
```

Изменим формулировку задачи. Пусть требуется найти все способы расстановки ферзей на шахматной доске $N \times N$. Для доски 8×8 ответ 92.

Возможный вариант реализации приведен ниже по тексту.

```
Procedure Solve(i:Integer);
 Var j:Integer;
 Begin
  If i<=N
    Then
      Begin
      For j:=1 To N Do
```

```
      If D_hod(i,j) Then
        Begin
         Hod(i,j);
         Solve(i+1);
         O_hod(i,j);
        End;
    End
     Else
      Begin
       Inc(S);{*Счетчик числа решений, глобальная
                переменная.*}
       Print;{*Вывод решения.*}
      End;
End;
```

Продолжим изучение задачи. Нас интересуют только несимметричные решения. Для доски 8×8 ответ 12. Это задание требует предварительных разъяснений. Из каждого решения задачи о ферзях можно получить ряд других при помощи вращений доски на 90°, 180° и 270° и зеркальных отражений относительно линий, разделяющих доску пополам (система координат фиксирована). Доказано, что в общем случае для любой допустимой расстановки N ферзей возможны три ситуации:

- при одном отражении доски возникает новая расстановка ферзей, а при поворотах и других отражениях новых решений не получается;
- поворот на 90° и его отражение дают еще две расстановки ферзей;
- три поворота и четыре отражения дают новые расстановки.

Для отсечения симметричных решений на всем множестве решений требуется определить некоторое отношение порядка. Представим решение в виде вектора длиной N, координатами которого являются числа от 1 до N. Для ферзя, стоящего в i-й строке, координатой его столбца является i-я координата вектора. Для того, чтобы не учитывать симметричные решения, будем определять минимальный вектор среди всех векторов, получаемых в результате симметрий. Процедуры *Sim1*, *Sim2*, *Sim3* выполняют зеркальные отображения вектора решения относительно горизонтальной, вертикальной и одной из диагональных осей. Известно (из геометрии), что композиция этих симметрий дает все определенные выше симметрии шахматной доски, причем не принципиально, в какой последовательности

выполняются эти процедуры для каждой конкретной композиции. Проверка «на минимальность» решения производится функцией *Cmp*, которая возвращает значение *True* в том случае, когда одно из симметричных решений строго меньше текущего решения.

Возможный вариант реализации (отсечение на выводе решений) приведен ниже по тексту.

```
Type TArray=Array[1..N] Of Integer;
....
Procedure Sim1(Var X:TArray);
 Var i:Integer;
 Begin
  For i:=1 To N Do X[i]:=N-X[i]+1;
 End;
Procedure Sim2(Var X:TArray);
 Var i,r:Integer;
 Begin
   For i:=1 To N Div 2 Do
     Begin
       r:=X[i]; X[i]:=X[N-i+1];X[N-i+1]:=r;
     End;
 End;
Procedure Sim3(Var X:TArray);
 Var Y:TArray;
     i:Integer;
 Begin
  For i:=1 To N Do Y[X[i]]:=i;
  X:=Y;
 End;
Function Cmp(X,Y:TArray):Boolean;
 Var i:Integer;
 Begin
  i:=1;
  While (i<=N) And (Y[i]=X[i]) Do Inc(i);
  If i>N
   Then Cmp:=False
   Else
     If Y[i]<X[i] Then Cmp:=True Else Cmp:=False;
 End;
Procedure Solve(i:Integer);
 Var j:Integer;f:Boolean;
     Y:TArray;
```

```
Begin
 If i<=N
  Then
   Begin
    For j:=1 To N Do
     If D_hod(i,j) Then
      Begin
          Hod(i,j);
          Solve(i+1);
          O_hod(i,j);
      End;
   End
  Else
    Begin
     f:=True;
     For j:=0 To 7 Do
      Begin
       Y:=X;
       If j And 1 =0 Then Sim1(Y);
       If j And 2 =0 Then Sim2(Y);
       If j And 4 =0 Then Sim3(Y);
       If Cmp(Y,X) Then f:=False;
      End;
     If f Then
      Begin
       Inc(S);{*Счетчик числа решений, глобальная
                переменная.*}
       Print;{*Вывод решения.*}
      End;
    End;
End;
```

Примечание

Программу решения задачи о шахматных ферзях можно написать по-разному. Ниже по тексту приведено одно из возможных решений*. Решение очень компактное, по-своему изящное, но не будем комментировать его. Попробуйте разобраться с ним и ответить на вопрос: в чем заключается отличие в стилях при написании программ в этом примере и в целом в данном учебнике.

---

* Текст программы из учебника по информатике для студентов педагогических вузов.

```
Program Ferz;
 Uses Crt;
 Const N=8;
 Var B:Array[1..N] Of Integer;
  Procedure Rf(i:Integer);
   Var j,k,p,t:Integer;
   Begin
    For j:=1 To N Do
     Begin
      B[i]:=j;k:=1;p:=0;
      While (k<1) And (p=0) Do
       Begin
        If (B[k]=B[i]) Or (Abs(k-i)=Abs(B[k]-B[i]))
             Then p:=1;
        Inc(k);
       End;
       If p=0 Then
        If i<N
         Then Rf(i+1)
         Else
          Begin
           For t:=1 To N Do Write(B[t]:3);
           WriteLn;
          End;
     End;
   End;
 Begin
  ClrScr;
  Rf(1);
 End.
```

2. «Задача о шахматном коне».

Существуют способы обойти шахматным конем доску, побывав на каждом поле по одному разу. Составить программу подсчета числа способов обхода.

Разбор задачи начинается с оценки числа 64! — таково общее число способов разметки доски 8×8. Каждую разметку следует оценивать на предмет того, является ли она способом обхода конем доски (решение в «лоб»). Затем оцениваем порядок сокращения перебора исходя из условия — алгоритм выбора очередного хода коня. Будем считать, что поля для хода выбираются по часовой стрелке. Объяснение иллюстрируется следующими рисунком 3.3.

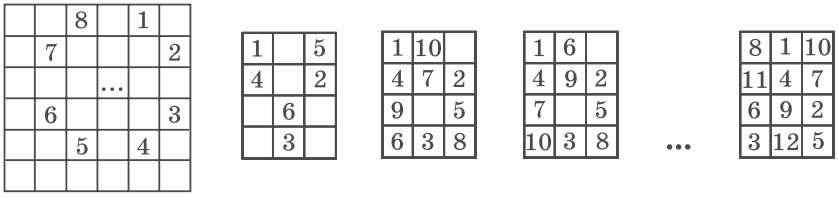


Рис. 3.3

Структуры данных.

```
Const N= ; M= ;
Dx:Array[1..8] Of Integer=(-2,-1,1,2,2,1,-1-2);
Dy:Array[1..8] Of Integer=(1,2,2,1,-1,-2,-2,-1);
Var A:Array[-1..N+2,-1..M+2] Of Integer;
    t:Integer;
```

Основной фрагмент реализации — процедура *Solve*.

```
 Procedure Solve(x,y,q:Integer);
  Var z,i,j:Integer;
  Begin
   A[x,y]:=q;
   If q=N*M
    Then Inc(t)
    Else
     For z:=1 To 8 Do
      Begin
             i:=x+Dx[z];j:=y+Dy[z];
             If A[i,j]=0 Then Solve(i,j,q+1)
      End;
   A[x,y]:=0;
  End;
```

Часть текста из основной программы имеет вид:

```
...
For i:=-1 To N+2 Do
 For j:=-1 To M+2 Do A[i,j]:=-1;{*Записываем
   "барьерные" элементы, тем самым исключая лишние
    операторы If, утяжеляющие текст программы.*}
For i:=1 To N Do
 For j:=1 To M Do A[i,j]:=0;
t:=0;
```

```
For i:=1 To N Do
 For j:=1 To M Do Solve(i,j,1);
WriteLn('число способов обхода конем доски ',
       N,'*',M,' — ',t);
....
```

Если попытаться составить таблицу из числа способов обхода конем шахматной доски для значений N и M, то окажется, что быстродействия вашего компьютера не хватит для решения этой задачи (а вы им так гордились!).

Изменим решение задачи о шахматном коне так, чтобы находился только один вариант обхода конем доски. Применим обход доски, основанный на правиле Варнсдорфа выбора очередного хода (предложено более 150 лет тому назад). Суть последнего в том, что при обходе шахматной доски коня следует ставить на поле, из которого он может сделать минимальное количество перемещений на еще не занятые поля, если таких полей несколько, можно выбирать любое из них. В этом случае в первую очередь занимаются угловые поля и количество «возвратов» в алгоритме значительно уменьшается. Найдите, для каких значений N и M ваш компьютер будет справляться с этой задачей?

Вариант процедуры *Solve* для этого случая.

```
Procedure Solve(x,y,q: Integer);
 Var   W: Array[1..8] Of Integer;
       xn,yn,i,j,m,min: Integer;
 Begin
  A[x,y]:=q;
  If (q<N*M) Then
    Begin
      For i:=1 To 8 Do
        Begin{*Формирование массива W.*}
          W[i]:=0;xn:=x+Dx[i];yn:=y+Dy[i];
          If (A[xn,yn]=0)
            Then
              Begin
                For j:=1 To 8 Do
                  If (A[xn+Dx[j],yn+Dy[j]]=0)
                    Then Inc(W[i]);
              End
            Else W[i]:=-1;
        End;
      i:=1;
      While (i<=8) Do
```

```
  Begin
    min:=Maxint;
    m:=1;{*Ищем клетку, из которой можно сделать
          наименьшее число перемещений.*}
    For j:=2 To 8 Do
      If W[j]<min Then
        Begin m:=j;min:=W[j];End;
    If (W[m]>=0) And (W[m]<Maxint) Then
        Begin
          Solve(x+Dx[m],y+Dy[m],l+1);
          W[m]:=Maxint;{*Отмечаем использованную
                        в переборе клетку.*}
        End;
    Inc(i);
  End;
End
Else
 Begin
   <вывод решения>;
   halt;{*Исключение этого оператора, нарушающего
         структурный стиль процедуры, одна из
         требуемых модификаций программы.*}
 End;
 A[x,y]:=0;
End;
```

3. «Задача о лабиринте».

Классическая задача для изучения темы. Как и предыдущие задачи, ее не обходят вниманием в любой книге по информатике. Формулировка проста. Дано клеточное поле, часть клеток занята препятствиями. Необходимо попасть из некоторой заданной клетки в другую заданную клетку путем последовательного перемещения по клеткам. Изложение задачи опирается на рисунок произвольного лабиринта и «прорисовку» действий по схеме простого перебора.

Классический перебор выполняется по правилам, предложенным в 1891 году Э. Люка в «Математических развлечениях»:

- в каждой клетке выбирается еще не исследованный путь;
- если из исследуемой в данный момент клетки нет путей, то возвращаемся на один шаг назад (в предыдущую клетку) и пытаемся выбрать другой путь.

Э. Люка в 1891 году сформулировал общую схему решения задач методом перебора вариантов.

Решение первой задачи.

```
Program Labirint;
 Const Nmax=...;
 Dx:Array[1..4] Of Integer=(1,0,-1,0);
 Dy:Array[1..4] Of Integer=(0,1,0,-1);
 Type MyArray=Array[0..Nmax+1,0..Nmax+1]
    Of Integer;
     Var A:MyArray;
         xn,yn,xk,yk,N:Integer;
 Procedure Init;
 ...
  Begin
 {*Ввод лабиринта, координат начальной и конечной
   клеток. Границы поля отмечаются как
   препятствия.*};
  End;
 Procedure Print;
 ....
  Begin
  {*Вывод матрицы A - метками выделен путь выхода
    из лабиринта.*};
  End;
 Procedure Solve(x,y,k:Integer);
{*k - номер шага,
   x,y - координаты клетки.*}
  Var i:Integer;
  Begin
   A[x,y]:=k;
   If (x=xk) And (y=yk)
    Then Print
    Else
      For i:=1 To 4 Do
        If A[x+Dx[i],y+Dy[i]]=0
          Then Solve(x+Dx[i],y+Dy[i],k+1);
   A[x,y]:=0;
  End;
Begin
 Init;
 Solve(xn,yn,1);
End.
```

С помощью данной схемы находятся все возможные варианты выхода из лабиринта. Их очередность (массивы констант *Dx*, *Dy*) определяется правилом просмотра соседних клеток. Если требуется найти один выход из лабиринта, то требуется, естественно, изменить решение.

Проделайте работу, сохранив при этом структурный вид программного кода.

Второй способ решения задачи о лабиринте (поиск кратчайшего по количеству перемещений пути) называют «методом волны».

Изложим суть метода. Начальную клетку помечают, например, меткой со значением 1, а затем значение метки увеличивается на единицу на каждом шаге (см. рисунок 3.4). Очередное значение метки записывается в свободные клетки, соседние с клетками, имеющими предыдущую метку. В любой момент времени множество клеток, не занятых препятствиями, разбивается на три класса: помеченные и изученные; помеченные и неизученные и непомеченные. Для хранения клеток второго класса лучше всего использовать структуру данных «очередь» (мы предполагаем, что она изучена ранее, например, по книге автора «Основы программирования»). Процесс заканчивается при достижении клетки выхода из лабиринта (есть путь) или при невозможности записать очередное значение метки (тупик). На рисунке 3.4 иллюстрируется этот процесс. Темные клетки означают препятствия. Начальная клетка находится вверху слева, конечная — внизу справа. Кроме того, из рассмотрения примеров должно следовать понимание того факта, что выход из лабиринта имеет кратчайшую длину.

1	■	9	8	■	10
2	3	■	7	8	9
3	4	5	6	7	■
4	■	6	■	8	9
5	6	7	■	9	10

Рис. 3.4

```
Program Labirint;
 Const NMax=…;
       MMax=…;
       Dx:Array[1..4] Of Integer=(-1,0,1,0);
       Dy:Array[1..4] Of Integer=(0,1,0,-1);
 Type MyArray=Array[0..NMax+1,0..MMax+1] Of Integer;
 Var A:MyArray;{*Глобальные переменные.*}
       N,M:Integer;
       xn,yn,xk,yk:Integer;
  Function Solve:Boolean;
   Type Och=Array[1..NMax*MMax,1..2] Of Integer;
```

```
Var t,i,j,ykr,ykw:Integer;
    O:Och;{*Очередь.*}
    Y:Boolean;
Begin
 A[xn,yn]:=1; ykr:=0{*Указатель чтения из
      очереди.*};
 ykw:=1; {*Указатель записи в очередь.*}
 Y:=False;{*Начальное присвоение.*}
 O[ykw,1]:=xn;O[ykw,2]:=yn;{*Заносим координаты
       начальной клетки в очередь.*}
 While (ykr<ykw) And Not(Y) Do
   Begin
   {*Пока очередь не пуста и не найдено решение.*}
   Inc(ykr);i:=O[ykr,1];j:=O[ykr,2];{*По
        указателю чтения берем элемент из
        очереди.*}
     If (i=xk) And (j=yk)
       Then Y:=True
        {*Если это клетка выхода из лабиринта, то
        решение найдено.*}
       Else
         For t:=1 To 4 Do
           If A[i+Dx[t],j+Dy[t]]=0 Then
             Begin
               {*В этой клетке мы еще не были.*}
               A[i+Dx[t],j+Dy[t]]:=A[i,j]+1;
               {*Присваиваем значение метки.*}
               Inc(ykw);O[ykw,1]:=i+Dx[t];
               O[ykw,2]:=j+Dy[t];{*Записываем
               координаты клетки в очередь.*}
             End;
   End;
 Solve:=Y;{*Результат работы функции Solve.*}
End;
```

Основная программа имеет вид:

```
Begin
 Init;{*Ввод лабиринта, координат начальной
       и конечной клеток. Текст процедуры
       не приводится.*}
 Assign(Output,'Output.txt');Rewrite(Output);
      {*Выводим результат (массив A) в файл.*}
 If Solve
```

```
  Then Print{*Текст процедуры Print в силу
       ее очевидности не приводится.*}
  Else WriteLn('Нет пути');
 Close(Output);
End.
```

Модифицируйте предыдущее решение так, чтобы находился не один путь, а, например, t путей, если они существуют.

4. «Задача о рюкзаке (перебор вариантов)».

В рюкзак загружаются предметы N различных типов (количество предметов каждого типа не ограничено). Максимальный вес рюкзака W. Каждый предмет типа i имеет вес w_i и стоимость v_i (i=1,2, ..., N). Требуется определить максимальную стоимость груза, вес которого равен W.

Обозначим количество предметов типа i через k_i, тогда требуется максимизировать

$$v_1 \times k_1 + v_2 \times k_2 + \ldots + v_N \times k_N$$

при ограничениях

$$w_1 \times k_1 + w_2 \times k_2 + \ldots + w_N \times k_N = W,$$

где k_i — целые ($0 \le k_i \le [W/w_i]$), квадратные скобки означают целую часть числа.

Рассмотрим простой переборный вариант решения задачи, работоспособный только для небольших значений N (определить, для каких?). Итак, данные:

```
Const MaxN=...;
Var   N,W:Integer;
{*Количество предметов, максимальный вес.*}
     Weight,Price:Array[1..MaxN] Of Integer;
{*Вес, стоимость предметов.*}
     Best,Now:Array[1..MaxN] Of Integer;
                 {*Наилучшее, текущее решения.*}
     MaxPrice:LongInt;{*Наибольшая стоимость.*}
```

Решение, его основная часть — процедура перебора:

```
Procedure Solve(k,w:Integer;st:LongInt);
{*k - порядковый номер группы предметов, w - вес,
       который следует набрать из оставшихся
       предметов, st - стоимость текущего решения.*}
 Var i:Integer;
 Begin
  If (k>N) And (st>MaxPrice)
```

```
Then
 Begin {*Найдено решение.*}
    Best:=Now;MaxPrice:=st;
 End
 Else If k<=N Then
   For i:=0 To W Div Weight[k] Do
     Begin
       Now[k]:=i;
       Solve(k+1,W-i*Weight[k],st+i*Price[k]);
     End;
 End;
```

Инициализация переменных, вывод решения и вызывающая часть (*Solve(*1,*W*,0*)*) очевидны.

5. «Задача о коммивояжере».

Классическая формулировка задачи известна уже более 200 лет: имеются N городов, расстояния между которыми заданы; коммивояжеру необходимо выйти из какого-то города, посетить остальные $N-1$ городов точно по одному разу и вернуться в исходный город. При этом маршрут коммивояжера должен быть минимальной длины (стоимости).

Задача коммивояжера принадлежит классу *NP*-полных т. е. неизвестны полиномиальные алгоритмы ее решения. В задаче с N городами необходимо рассмотреть $(N-1)!$ маршрутов, чтобы выбрать маршрут минимальной длины. Итак, при больших значениях N невозможно за разумное время получить результат.

Оказывается, что и при простом переборе не обязательно просматривать все варианты. Это реализуется в данной классической схеме реализации.

Структуры данных.

```
Const Max=…;
Var    A:Array[1..Max,1..Max] Of Integer;
{*Матрица расстояний между городами.*}
  B:Array[1..Max,1..Max] Of Byte;{*Вспомогательный
        массив, элементы каждой строки матрицы
       сортируются в порядке возрастания, но сами
       элементы не переставляются, а изменяются
       в матрице B номера столбцов матрицы A.*}
```

Пример. На рисунке 3.5 приведены матрицы A и B (после сортировки элементов каждой строки матрицы A).

A

@	27	43	16	30	26
7	@	16	1	30	25
20	13	@	35	5	0
21	16	25	@	18	18
12	46	27	48	@	5
23	5	5	9	5	@

B

4	6	2	5	3	1
4	1	3	6	5	2
6	5	2	1	4	3
2	5	6	1	3	4
6	1	3	2	4	5
2	3	5	4	1	6

Рис. 3.5

Примечание

Символом @ обозначено бесконечное расстояние.

```
Way,BestWay:Array[1..Max] Of Byte;{*Хранится
      текущее решение и лучшее решение.*}
Nnew:Array[1..Max] Of Boolean;{*Значение
      элемента массива False говорит о том,
      что в соответствующем городе коммивояжер
      уже побывал.*}
BestCost:Integer;{*Стоимость лучшего решения.*}
```

Идея решения. Пусть мы находимся в городе с номером v. Наши действия.

Шаг 1. Если расстояние (стоимость), пройденное коммивояжером до города с номером v, не меньше стоимости найденного ранее наилучшего решения (*BestCost*), то следует выйти из данной ветви дерева перебора.

Шаг 2. Если рассматривается последний город маршрута (осталось только вернуться в первый город), то следует сравнить стоимость найденного нового решения со стоимостью лучшего из ранее найденных решений. Если результат сравнения положительный, то значения *BestCost* и *BestWay* следует изменить и выйти из этой ветви дерева перебора.

Шаг 3. Пометить город с номером v как рассмотренный, записать этот номер по значению *Count* в массив *Way*.

Шаг 4. Рассмотреть пути коммивояжера из города v в ранее не рассмотренные города. Если такие города есть, то перейти на эту же логику с измененными значениями v, *Count*, *Cost*, иначе на следующий шаг.

Шаг 5. Пометить город v как нерассмотренный и выйти из данной ветви дерева перебора.

Прежде чем перейти к обсуждению алгоритма, целесообразно разобрать этот перебор на примере. На рисунке 3.6 приведен пример и порядок просмотра городов. В кружках указаны но-

мера городов, рядом с кружками — суммарная стоимость проезда до этого города из первого.

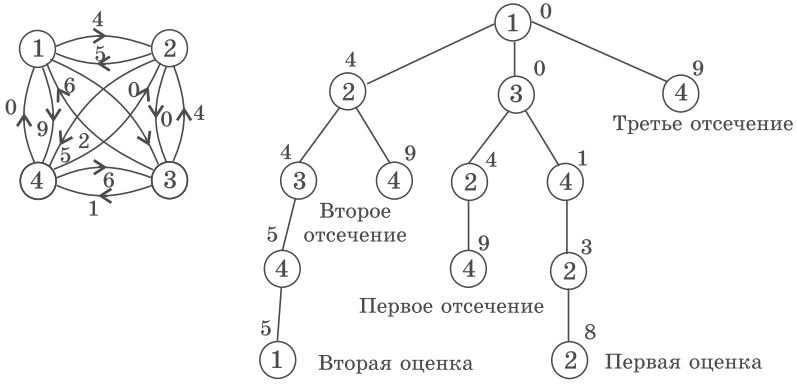


Рис. 3.6

Итак, запись алгоритма.

```
Procedure Solve(v,Count:Byte;Cost:Integer);
{*v - номер текущего города; Count - счетчик числа
 пройденных городов; Cost - стоимость текущего
 решения.*}
 Var i:Integer;
 Begin
  If Cost>BestCost Then Exit;{*Стоимость текущего
         решения превышает стоимость лучшего из
         ранее полученных.*}
  If Count=N Then
    Begin
     Cost:=Cost+A[v,1];Way[N]:=v;{*Последний город
          пути. Добавляем к решению стоимость
          перемещения в первый город и сравниваем
          его с лучшим из ранее полученных.*}
     If Cost<BestCost Then
       Begin BestCost:=Cost; BestWay:=Way;End;
     Exit;{*Оператор нарушает структурный стиль
          программирования - "любой фрагмент логики
          должен иметь одну точку входа и одну точку
          выхода. Следует убрать его" .*}
    End;
```

```
    Nnew[v]:=False;Way[Count]:=v;{*Город с номером v
          пройден, записываем его номер в путь
          коммивояжера.*}
    For i:=1 To N Do
     If Nnew[B[v,i]] Then
        Solve(B[v,i], Count+1,Cost+A[v,B[v,i]]);
          {*Поиск города, в который коммивояжер
          может пойти из города с номером v.*}
     Nnew[v]:=True; {*Возвращаем город с номером v
          в число непройденных.*}
    End;
```

Первый вызов — *Solve(*1,1,0*)*. Разработка по данному «шаблону» работоспособной программы — техническая работа. Если вы решитесь на это, то есть смысл проверить ее работоспособность на примере матрицы A, приведенной на рисунке 3.7. Решение имеет вид 1 8 9 2 5 6 10 7 4 3 1, его стоимость 158.

@	32	19	33	22	41	18	15	16	31
32	@	51	58	27	42	35	18	17	34
19	51	@	23	35	49	26	34	35	41
33	58	23	@	33	37	23	46	46	32
22	27	35	33	@	19	10	23	23	9
41	42	49	37	19	@	24	42	42	10
18	35	26	23	10	24	@	25	25	14
15	18	34	46	23	42	25	@	1	32
16	17	35	46	23	42	25	1	@	32
31	34	41	32	9	10	14	32	32	@

Рис. 3.7

Задание. Оцените время работы программы. Если у вас мощный компьютер, то создайте матрицу A[1..50,1..50] и попытайтесь найти наилучшее решение с помощью разобранного метода. Заметим, что набор 2500 чисел — утомительное занятие и разумная лень — «двигатель прогресса».

3.3. Динамическое программирование

Идея метода. Это важнейший раздел изучаемой темы, поэтому вполне применим принцип: чем больше разнообразных задач, тем лучше. Разбор основной идеи метода и ее обсуждение можно сделать на *классической задаче о Черепашке*.

Черепашке необходимо попасть из пункта A в пункт B. На каждом углу она может поворачивать только на север или только на восток. Время движения по каждой улице указано на рисунке 3.8. Требуется найти минимальное время, за которое Черепашка может попасть из пункта A в пункт B.

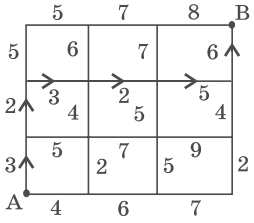


Рис. 3.8

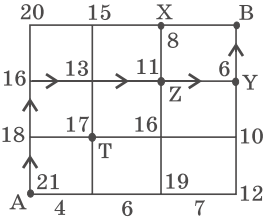


Рис. 3.9

Путь, показанный на рисунке 3.8 линиями со стрелкой, требует 21 единицу времени. Отметим, что каждый путь состоит из 6 шагов: трех на север и трех на восток. Количество возможных путей Черепашки 20 (C_6^3).

Мы можем перебрать все пути и выбрать самый быстрый. Это метод полного перебора (ответ — 21). Для вычисления каждого времени требуется пять операций сложения, а для нахождения ответа — 100 операций сложения и 19 операций сравнения. Задача решается вручную. Однако при N, равном 8, у Черепашки уже 12870 путей. Подсчет времени для каждого пути требует 15 операций сложения, а в целом — 193050 сложений и 12869 сравнений, то есть порядка 200000 операций. Компьютер при быстродействии 1000000 операций в секунду найдет решение за 0.2 секунды, а человек?

Предположим, что N равно 30, тогда количество различных путей $60!\times(30!\times30!)$. Это очень большое число, его порядок 10^{17}. Количество операций, необходимых для поиска решения, равно 60×10^{17}. Компьютер с быстродействием 1000000 операций в секунду выполнит за год порядка 3.2×10^{13} операций (подсчитайте). А сейчас нетрудно прикинуть количество лет, требуемых для решения задачи.

Вернемся к исходной задаче. Начнем строить путь Черепашки от пункта B. Каждому углу присвоим вес, равный минимальному времени движения Черепашки от этого угла до пункта B. Как его находить? От углов X, Y решение очевидно. Для угла Z время движения Черепашки в пункт B через угол X равно 15 единицам, а через угол Y — 11 единицам. Берем минима-

льное значение, т. е. вес угла будет равен 11. Продолжим вычисления. Их результаты приведены на рисунке 3.9. Путь, отмеченный стрелками, является ответом задачи. Оценим количество вычислений. Для каждого угла необходимо выполнить не более двух операций сложения и одной операции сравнения, т. е. три операции. При N, равном 300, количество операций — $3 \times 301 \times 301$, это меньше 1000000, и компьютеру потребуется меньше одной секунды. Итак, много лет при $N=30$ и 1 секунда при $N=300$.

Идея второго способа решения задачи основана на методе динамического программирования. Заслуга его открытия принадлежит американскому математику Ричарду Беллману. Выделим особенность задачи, которая позволяет решить ее описанным способом. Мы начинаем с угла B (см. рисунок 3.9), и для каждого угла найденное число является решением задачи меньшей размерности. Поясним эту мысль. Если бы мы решали задачу поиска пути Черепашки из пункта T в пункт B, то найденное число 17 — решение задачи. Для каждого угла найденное значение времени не изменяется и может быть использовано на следующих этапах.

3.4. Примеры задач для разбора идеи метода динамического программирования

1. «Треугольник».

На рисунке 3.10 изображен треугольник из чисел. Напишите программу, которая вычисляет наибольшую сумму чисел, расположенных на пути, начинающемся в верхней точке треугольника и заканчивающемся на основании треугольника.

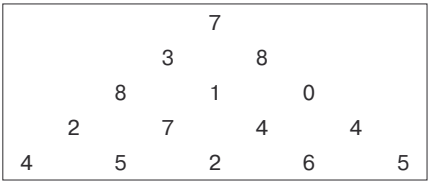


Рис. 3.10

- Каждый шаг на пути может осуществляться вниз по диагонали влево или вниз по диагонали вправо.
- Число строк в треугольнике больше 1 и меньше 100.
- Треугольник составлен из целых чисел от 0 до 99.

7	0	0	0	0
3	8	0	0	0
8	1	0	0	0
2	7	4	4	0

Рис. 3.11

```
0 7
0 10 15
0 18 16 15
0 20 25 20 19
0 24 30 27 26 24
```

Рис. 3.12

Рассмотрим идею решения на примере, приведенном в тексте задачи. Входные данные запишем в матрицу *D*. Значения элементов матрицы *D* приведены на рисунке 3.11

Будем вычислять матрицу *R:array[1..MaxN,0..MaxN]* следующим образом, предварительно обнулив ее элементы:

```
R[1,1]:=D[1,1];
For i:=2 To N Do
 For j:=1 To i Do
   R[i,j]=max(D[i,j]+R[i-1,j],D[i,j]+R[i-1,j-1]);
```

где *max* — функция вычисления максимального из двух чисел.

Осталось найти наибольшее значение в последней строке матрицы *R*, оно равно 30 (см. рисунок 3.12).

2. «Степень числа».

Даны два натуральных числа N и k. Требуется определить выражение, которое вычисляет k^N. Разрешается использовать операции умножения и возведения в степень, круглые скобки и переменную с именем k. Умножение считается одной операцией, возведению в степень q соответствует $q-1$ операция. Найти минимальное количество операций, необходимое для возведения в степень N. Желательно сделать это для как можно больших значений N.

Так, при $N=5$ требуются три операции — $(k\times k)^2\times k$.

Определим массив *Op*, его элемент *Op*[*i*] предназначен для хранения минимального количества операций при возведении в степень i (*Op*[1]=0). Для вычисления выражения, дающего N-ю степень числа k, арифметические операции применяют в некоторой последовательности, согласно приоритетам и расставленным скобкам. Рассмотрим последнюю примененную операцию.

Если это было умножение, тогда сомножителями являются натуральные степени числа k, которые в сумме дают N. Сте-

пень каждого из сомножителей меньше N, и ранее вычислено, за какое минимальное число операций ее можно получить. Итак:

```
opn1:=Min{по всем p:1≤p<N, Op[p]+Op[n-p]+1}.
```

Если последней операцией являлось возведение в степень, то

```
opn2:=Min;{ для всех p (≠1) - делителей N,
  Op[N div p]+p-1}.
```

Результат — *Op*[*N*]=*Min*(*opn1*,*opn2*). Фрагмент реализации:

```
Const MaxN=1000;
Var   N, k: Integer;
Op: Array[1..MaxN] Of Integer;
Procedure Solve;
 Var i, j: Integer;
 Begin
  For i:=2 To N Do
   Begin
    Op[i]:=Op[i-1]+1;
    For j:=2 To i-1 Do
     Begin
      Op[i]:=Min(Op[i], Op[j]+Op[i-j]+1);
      If i Mod j=0 Then
       Op[i]:=Min(Op[i], Op[i Div j]+j-1);
     End;
   End;
 End;
```

3. «Алгоритм Нудельмана–Вунша» (из молекулярной биологии).

Молекулы ДНК содержат генетическую информацию. Моделью ДНК можно считать длинное слово из четырех букв (А, Г, Ц, Т). Даны два слова (длины M и N), состоящие из букв А, Г, Ц, Т. Найти подпоследовательность наибольшей длины, входящую в то и другое слово.

Пусть есть слова ГЦАТАГГТЦ и АГЦААТГГТ. Схема решения иллюстрируется рисунком 3.13. На рисунке закрашены клетки, находящиеся на пересечении строки и столбца с одинаковыми буквами.

Принцип заполнения таблицы W следующий: элемент $W[i,j]$ равен наибольшему из чисел $W[i-1,j]$, $W[i,j-1]$, а если клетка $<i,j>$ закрашена, то и $W[i-1,j-1]+1$.

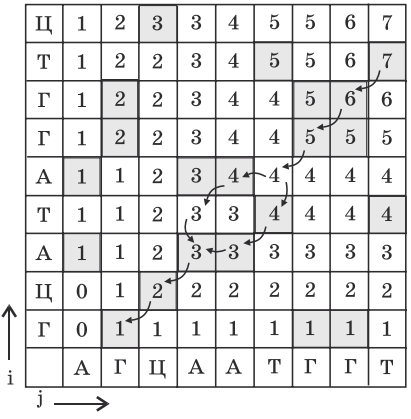


Рис. 3.13

Формирование первой строки и первого столбца выполняется до заполнения таблицы и осуществляется так: единицей отмечается первое совпадение, затем эта единица автоматически заносится во все оставшиеся клетки.

Например, *W*[3,1] — первое совпадение в столбце, затем эта единица идет по первому столбцу. Подпоследовательность формируется при обратном просмотре заполненной таблицы от клетки, помеченной максимальным значением. Путь — это клетки с метками, отличающимися на единицу, буквы выписываются из закрашенных клеток. Последовательность этих букв — ответ задачи. Для нашего примера две подпоследовательности: ГЦААГГТ и ГЦАТГГТ.

Фрагмент основного алгоритма.

```
...
For i:=1 To Length(S1) Do
 For j:=1 To Length(S2) Do
  Begin
   A[i,j]:=Max(A[i-1,j],A[i,j-1]);
   If S1[i]=S2[j] Then
     A[i,j]:=Max(A[i,j], A[i-1,j-1]+1);
  End;
WriteLn('Ответ: ',A[Length(S1),Length(S2)]);
....
```

4. «Разбиение выпуклого N-угольника».

Дан выпуклый *N*-угольник, заданный координатами своих вершин в порядке обхода. Он разрезается *N*-2 диагоналями на

треугольники. Стоимость разрезания определяется суммой длин всех использованных диагоналей. Найти разрез минимальной стоимости.

Идея решения разбирается с использованием рисунка 3.14.

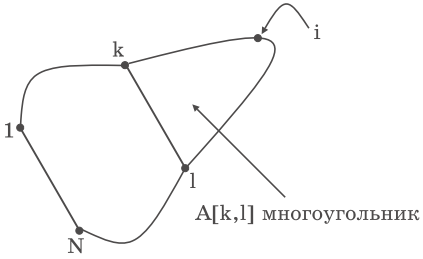


Рис. 3.14

Обозначим через $S[k,l]$ стоимость разрезания многоугольника $A[k,l]$ диагоналями на треугольники. При $l=k+1$ или $k+2$ $S[k,l]=0$, следовательно, $l>k+2$. Вершина с номером i, i изменяется от $k+1$ до $l-1$, определяет какое-то разрезание многоугольника $A[k,l]$. Стоимость разрезания определим как: $S[k,l]=min\{$*длина диагонали* $<k,i>+$*длина диагонали* $<i,l>+S[k,i]+S[i,l]\}$. При этом следует учитывать, что при $i=k+1$ диагональ $<k,i>$ является стороной многоугольника и ее длина считается равной нулю. Аналогично при $i=l-1$.

5. «Задача о камнях».

Из камней весом $p_1, p_2, ..., p_N$ набрать вес W или, если это невозможно, максимально близкий к W снизу вес.

Рассмотрим *идею решения* на следующих данных:

N равно 5, $p_1=5$, $p_2=7$, $p_3=9$, $p_4=11$, $p_5=13$, $W=19$.

Сформируем матрицу A (*A:Array*[*1..N,0..W*] *Of Integer*), в которой номер строки соответствует номеру камня, а номер столбца — набираемому весу. В нулевой столбец запишем нули, в первую строку (по столбцам) до веса первого камня нули, а затем вес первого камня (пытаемся набрать требуемый вес одним камнем). Во второй строке фиксируем результат набора веса с помощью двух камней и т. д. После заполнения A имеет вид, показанный на рисунке 3.15. Вес 19 набрать нельзя, ближайший снизу вес 18.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	*0*	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
2	*0*	0	0	0	0	5	5	*7*	7	7	7	7	12	12	12	12	12	12	12	12
3	0	0	0	0	0	5	5	*7*	7	9	9	9	12	12	14	14	16	16	16	16
4	0	0	0	0	0	5	5	*7*	7	9	9	11	12	12	14	14	16	16	18	*18*
5	0	0	0	0	0	5	5	7	7	9	9	11	12	12	14	14	16	16	18	*18*

Рис. 3.15

Для заполнения текущей строки достаточно знать только предыдущую строку (результат решения задачи меньшей на единицу размерности), а в элементе массива $A[i,j]$ записывается решение задачи о камнях при $N=i$ и $W=j$.

Программный код решения компактен и не требует пояснений.

```
Procedure Solve;
 Var i,j:Integer;
 Begin
  For i:=2 To N Do
   For j:=1 To W Do
     Begin
      If j-P[i]>=0
      Then
        A[i,j]:= Max(A[i-1,j],
        A[i-1,j-P[i]]+P[i])
      Else A[i,j]:=A[i-1,j];
     End;
 End;
```

Для ответа на вопрос, «из каких камней набирается вес», решение необходимо дополнить простой рекурсивной процедурой. В таблице на рисунке 3.15 курсивом выделена часть элементов, задействованных при работе процедуры, а жирным шрифтом те значения i и j, при которых осуществляется вывод веса камня $P[i]$.

```
Procedure Way(i,j:Integer);
 Begin
  If (i=1) And (A[i,j]=0)
   Then Exit
   Else
    If i=1
```

```
    Then
     Begin
      Way(i,j-P[i]);
      Write(P[i],' ');
     End
    Else
     If A[i,j]<>A[i-1,j]
      Then
       Begin Way(i,j-P[i]);Write(P[i],' ');End
      Else Way(i-1,j);
End;
```

6. «Задача о рюкзаке (к весу камней добавляется и их стоимость)».

Напомним формулировку задачи. В рюкзак загружаются предметы N различных типов (количество предметов каждого типа не ограничено). Вес рюкзака должен быть меньше или равен W. Каждый предмет типа i имеет вес w_i и стоимость v_i (i=1,2, …, N). Требуется определить максимальную стоимость груза, вес которого не превышает W.

Рассмотрим случай, когда вес рюкзака в точности равен W. Обозначим количество предметов типа i через k_i, тогда требуется максимизировать $v_1 \times k_1 + v_2 \times k_2 + \ldots + v_N \times k_N$ при ограничениях

$$w_1 \times k_1 + w_2 \times k_2 + \ldots + w_N \times k_N = W,$$

где k_i — целые ($0 \le k_i \le [W/w_i]$), квадратные скобки означают целую часть числа.

Формализуем задачу следующим образом. Шаг i ставится в соответствие типу предмета i=1,2,…,N. Состояние y_i на шаге i выражает суммарный вес предметов, решение о загрузке которых принято на шагах 0,1,…,i. При этом $y_n=W$, y_i=0,1,…,W при i=1,2,…,N−1. *Варианты решения* k_i на шаге i описываются количеством предметов типа i, $0 \le k_i \le [W/w_i]$.

Уточним идею на конкретных данных. Пусть W=6, и дано четыре предмета (см. таблицу 3.1).

Таблица 3.1

i	w_i	v_i
1	2	50
2	3	90
3	1	30
4	4	140

Схема работы для данного примера приведена на рисунке 3.16. В кружочках выделены только достижимые состояния (суммарные веса для каждого шага в соответствии с приведенной выше формализацией) на каждом шаге. В круглых скобках указаны стоимости соответствующих выборов, в квадратных скобках — максимальная стоимость данного заполнения рюкзака. «Жирными» линиями выделен способ наилучшей загрузки рюкзака.

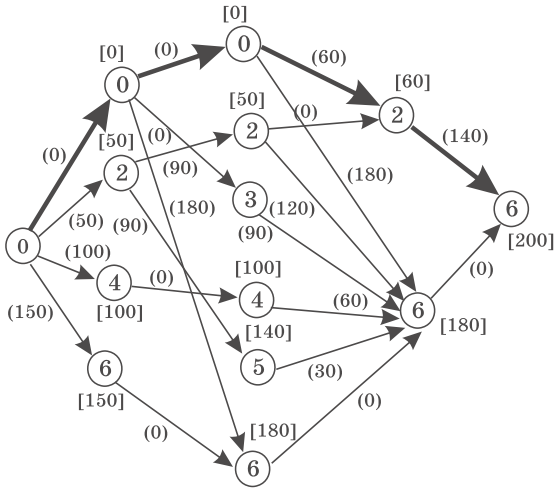


Рис. 3.16

Текст решения.

```
Const MaxN=...;
      MaxK=...;
Type Thing=Record
           W,V: Word
     End;
Var   A: Array[1..MaxN,0..MaxK] Of Word;
      P: Array[1..MaxN] Of Thing;
      Old,NewA:Array[0..MaxK] Of LongInt;
      N,W:Integer;
...
Procedure Solve;
 Var k,i,j:Integer;
 Begin
  FillChar(Old,SizeOf(Old),0);
  For k:=1 To N Do
```

```
  Begin{*Цикл по шагам.*}
   FillChar(NewA,SizeOf(NewA),0);
   For i:=0 To W Do {*Цикл по состояниям шага.*}
    For j:=0 To i Div P[k].W Do {*Цикл по вариантам
           решения - количеству предметов каждого
           вида.*}
     If j*P[k].V+Old[i-j*P[k].W]>=NewA[i] Then
      Begin
       NewA[i]:=j*P[k].V+Old[i-j*P[k].W];
       A[k,i]:=j;{*Здесь j количество предметов?*}
      End;
     Old:=NewA;
  End;
End;
```

Вывод наилучшего решения.

```
Procedure OutWay(k,l:Integer);
 Begin
  If k=0 Then Exit
        Else
          Begin
            OutWay(k-1,l-A[k,l]*P[k].W);{*A здесь
                                          вес.*}
            Write(A[k,l],' ');
          End;
 End;
```

Первый вызов — *OutWay(N,W)*. Эту схему реализации принято называть «прямой прогонкой». Ее можно изменить. Пусть пункт два формализации задачи звучит следующим образом. Состояние y_i на шаге i выражает суммарный вес предметов, решение о загрузке которых принято на шагах i, i+1, …, N при этом $y_1=W$, $y_i=0,1,\ldots,W$ при $i=2,3,\ldots,N$. В этой формулировке схему реализации называют «обратной прогонкой». Иллюстрация к этой схеме приведена на рисунке 3.17.

3.5. Метод ветвей и границ*

Идея метода ветвей и границ обычно излагается по работе Дж. Литтла. Не отступим и мы от этой традиции. Суть идеи проста. Все перебираемые варианты следует разбить на классы

---

* Литтл Дж., Мурти К., Суини Д., Кэрел Е. Я. Алгоритм решения задачи коммивояжера.//Экономика и математические методы. 1965. №1. С.13–22.

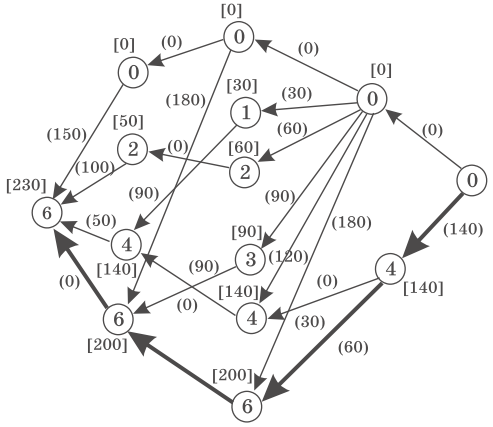


Рис. 3.17

(блоки), сделать оценку снизу (при минимизации) для всех решений из одного класса, и если она больше ранее полученной, то отбросить все варианты из этого класса. Весь вопрос в том, как разделять на классы.

Пусть решается задача о коммивояжере. Дана матрица расстояний — полный граф (см. рисунок 3.18).

	1	2	3	4	5	6
1	–	7	3	10	17	5
2	9	–	4	5	8	6
3	13	2	–	9	11	14
4	5	8	6	–	3	6
5	16	11	13	10	–	8
6	6	5	9	8	4	–

Рис. 3.18

Если мы будем вычитать одно и то же число из всех элементов строки или столбца, то суммарная стоимость пути коммивояжера не изменится. При вычитании константы из элементов строки стоимость любого пути уменьшится на эту константу, ибо числа в строке соответствуют выезду из города, а выезд из этого города обязательно присутствует в пути. Точно так же и вычитание из элементов столбца — въезд в город. Вычитаемая константа входит в оценку пути, но не рассматривается дальше, после изменения матрицы. Найдем минимум в каждой строке и вычтем его. Получается матрица,

которая приведена на рисунке 3.19. Сумма вычитаемых констант равна 24.

	1	2	3	4	5	6
1	–	4	0	7	14	2
2	5	–	0	1	4	2
3	11	0	–	7	9	12
4	2	5	3	–	0	3
5	8	3	5	2	–	0
6	2	1	5	4	0	–

Рис. 3.19

Аналогичную операцию выполним со столбцами: из элементов первого вычитаем 2, а из элементов четвертого — 1. Итоговая сумма — 27. Все пути коммивояжера уменьшились на это значение.

	1	2	3	4	5	6
1	–	4	*0*	6	14	2
2	3	–	0	*0*	4	2
3	9	*0*	–	6	9	12
4	0	5	3	–	*0*	3
5	6	3	5	1	–	*0*
6	0	1	5	3	0	–

Рис. 3.20

Если получается выбор по одному нулю в каждом столбце и строке (они выделены курсивом в матрице на рисунке 3.20), то тем самым мы получаем путь коммивояжера со стоимостью 27. Это путь (1→3→2→4→5→6→1) с минимальной стоимостью, любой другой путь имеет бóльшую стоимость. Значение 27 — это оценка снизу всех маршрутов коммивояжера. В данном случае она достигается на конкретном маршруте. Очевидно, что мы просто очень удачно подобрали пример. Не обязательно после приведения существование пути по ребрам с нулевой стоимостью. Продолжим рассмотрение. Пусть дана другая матрица расстояний, она приведена на рисунке 3.21.

	1	2	3	4	5	6
1	–	1	2	3	4	5
2	2	–	3	4	5	6
3	3	4	–	5	6	7
4	7	6	5	–	7	8
5	6	5	4	3	–	2
6	5	4	3	2	1	–

Рис. 3.21

После приведения по строкам и столбцам (сумма констант приведения равна 14) получаем новую матрицу, которая представлена на рисунке 3.22.

	1	2	3	4	5	6
1	–	0^1	0^0	1	3	4
2	0^0	–	0	1	3	4
3	0	1	–	1	3	4
4	4	3	1	–	1	0
5	4	3	1	0	–	0^0
6	4	3	1	0^0	0	–

Рис. 3.22

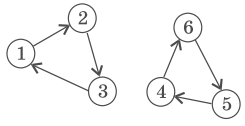


Рис. 3.23

Выделенные курсивом нули дают путь, но это не путь коммивояжера. Его вид показан на рисунке 3.23. Наши дальнейшие действия (шаг ветвления). Рассмотрим нули в приведенной матрице (см. рисунок 3.22), именно нуль в позиции (1, 2). Он, естественно, означает, что стоимость переезда из первого во второй город равна нулю. Однако если исключить (запретить) переезд из первого во второй город, то въезжать во второй город все равно придется. Цены въезда указаны во втором столбце — въезжать из третьего города дешевле всего. Так как выезжать из первого города нам когда-либо придется, то дешевле всего выехать в третий город — нулевая стоимость. Вычисляем сумму этих минимумов, она равна 1+0=1. Суть этой единицы в том, что если не ехать из первого города во второй, то придется заплатить не менее 1. Эта оценка нуля указана в матрице верхним индексом. Сделаем оценки всех нулей и выберем элемент с максимальной оценкой. Если таких нулей несколько,

то выбираем любой. Итак, в чем суть ветвления. Все маршруты коммивояжера разбиваются на два класса, содержащих ребро (1,2) и не содержащих ребро (1,2). Для последнего класса оценка снизу увеличивается на единицу, т. е. равна 15. Для маршрутов из первого класса продолжаем работать с матрицей. Исключаем (вычеркиваем) первую строку и второй столбец. Кроме того, в уменьшенной матрице на место (2,1) следует поставить прочерк, означающий невозможность соответствующего переезда. После этих манипуляций матрица примет следующий вид, приведенный на рисунке 3.24.

	1	3	4	5	6
2	–	0^2	1	3	4
3	0^5	–	1	3	4
4	4	1	–	1	0^1
5	4	1	0^0	–	0^0
6	4	1	0^0	0^1	–

Рис. 3.24

Продолжим наши действия по оценке маршрутов из первого класса (см. рисунки 3.25, 3.26, 3.27). Сокращенную матрицу, к сожалению, приводить нельзя — минимальные элементы во всех строках и столбцах равны нулю. Таким образом, оценка снизу для класса маршрутов, содержащих (1,2), остается равной 14. С сокращенной матрицей выполним аналогичные действия — оценим нули, выберем нуль, соответствующий переезду из третьего города в первый. Суть действий в разбивке этого класса маршрутов на подклассы по той же самой схеме — с (3,1) и без (3,1). Для второго подкласса оценка снизу 14+5=19. С первым подклассом аналогичные действия — исключаем столбец и строку, на место (2,3) в таблице пишем запрет на перемещение. Матрица допускает операцию приведения. Вычитаем единицу из элементов первой строки и первого столбца.

	3	4	5	6
2	–	1	3	4
4	1	–	1	0
5	1	0	–	–
6	1	0	0	1

Рис. 3.25

	3	4	5	6
2	–	0^2	2	3
4	0^0	–	1	0^0
5	0^0	0^0	–	0^0
6	0^0	0^0	0^1	–

Рис. 3.26

	3	5	6
4	–	1	0^1
5	0^0	–	0^0
6	0^0	0^0	–

Рис. 3.27

Нижняя оценка маршрутов этого подкласса возрастает на два, итого 14+2=16. Следующий шаг приводит к матрице еще меньшего размера. Весь процесс работы по рассматриваемой схеме представлен на рисунке 3.28. Вершины этого дерева представляют классы решений. Черта над цифрами означает, что соответствующий класс не содержит какой-то конкретный переезд из города в город. Числа у вершин дерева означают оценки снизу для маршрутов из данного класса. Итак, получена первая оценка — 16 для маршрута 1→2→4→6→5→3→1. Все подклассы решений, у которых оценка снизу больше или равна 16, исключаются из рассмотрения — очевидный факт после изложения схемы решения. Остался только подкласс без переезда из первого города во второй, его оценка равна 15. Однако после первого же ветвления получаются подклассы с оценками 16 и 17. Обработку можно заканчивать. Найден наилучший маршрут коммивояжера, стоимость проезда по этому маршруту равна 16.

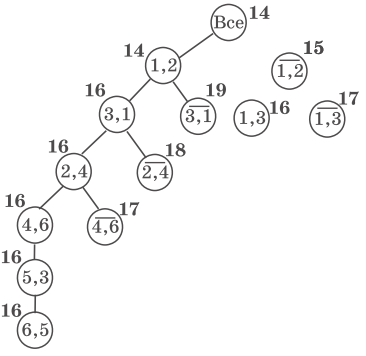


Рис. 3.28

Программная реализация метода ветвей и границ — хорошая проверка техники программирования (структурного стиля самого процесса написания программы). Что лучше? Или работать с матрицами различной размерности, или вводить дополнительные структуры для хранения исключенных номеров строк и столбцов? Вопросы можно продолжить. Ответ даст эксперимент с различными версиями программ реализации алгоритма.

В п. 3.2.2 рассмотрена одна схема решения задачи коммивояжера, в данном пункте — другая. В чем их отличие для различных значений размерности задачи N? По некоторым оценкам, с помощью метода ветвей и границ можно решать задачи с $N \le 100$.

3.6. Метод «решета»

Идея метода. Решето представляет собой метод комбинаторного программирования, который рассматривает конечное множество и исключает все элементы этого множества, не представляющие интереса. Он является логическим дополнением к процессу поиска с возвратом (*backtrack*), в котором мы пытаем-

ся постепенно перечислить все возможные варианты решения. В решете идут от обратного — от всего множества решений задачи.

Решето Эратосфена. Эратосфен — греческий математик, 275–194 гг. до нашей эры. Классическая задача поиска простых чисел в интервале [2..N] — инструмент для обсуждения метода. Объяснение строится на основе рисунка 3.29 и фрагмента алгоритма.

Исходная последовательность:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Выделенные элементы последовательности отбрасываются, в результате получаем:

2	3	5	6	7	9	10	11	13	15	17	19	21	23	25	27	29	31

2	3	5	6	7	9	10	11	13	15	17	19	21	23	25	27	29	31

Выделенные элементы последовательности отбрасываем, в результате получаем:

2	3	5	6	7	10	11	13	17	19	23	25	29	31

Рис. 3.29

```
Const N= ...;{*Верхняя граница интервала чисел.*}
Type Number=Set Of 2..N;{решето, N<256}
Var S:Number;
Procedure Search(Var S:Number);
 Var i,j:2..N;
 Begin
  S:=[2..N];
  For i:=2 To N Div 2 Do
   If i In S Then
     Begin
      j:=i+i;
      While j<=N Do
       Begin S:=S-[j];j:=j+i; End;
     End;
End;
```

Логическим развитием задачи является ее решение при N бóльших 256. В этом случае необходимо ввести массив с множественным типом данных элементов.

«Быки и коровы»[*].

Компьютер и ребенок играют в следующую игру. Ребенок загадывает последовательность из четырех (не обязательно различных) цветов, выбранных из шести заданных. Для удобства будем обозначать цвета цифрами от 1 до 6. Компьютер должен отгадать последовательность, используя информацию, которую он получает из ответов ребенка.

Компьютер отображает на экране последовательность, а ребенок должен ответить (используя для ввода ответа клавиатуру) на два вопроса:

- сколько правильных цветов на неправильных местах;
- сколько правильных цветов на правильных местах.

Пример. Предположим, что ребенок загадал последовательность 4655. На рисунке 3.30 приведена последовательность шагов по отгадыванию числа.

Компьтер	Ответ ребенка	
1234	1	0
5156	2	1
6165	2	1
5625	1	2
5653	1	2
4655	0	4

Рис. 3.30

Один из возможных способов отгадать последовательность такой:
написать программу, которая всегда отгадывает последовательность не более чем за шесть вопросов, в худшем случае — за десять.

Правильный выбор структур данных — это уже половина решения. Итак, структуры данных и процедура их инициализации.

```
Const Pmax=6*6*6*6;
Type Post=String[4];
Var A:Array[1..Pmax] Of Post;
    B:Array[1..Pmax] Of Boolean;
```

[*] Модификация задачи Антона Александровича Суханова.

```
      cnt:Integer;{*Счетчик числа ходов.*}
      ok:Boolean;{*Признак - решение найдено.*}
Procedure Init;
 Var i,j,k,l:Integer;
 Begin
  For i:=1 To 6 Do
   For j:=1 To 6 Do
    For k:=1 To 6 Do
     For l:=1 To 6 Do
      A[(i-1)*216+(j-1)*36+(k-1)*6+l]:=
        Chr(i+Ord('0'))+Chr(j+Ord('0'))+
            +Chr(k+Ord('0'))+Chr(l+Ord('0'));
   For i:=1 To Pmax Do B[i]:=True;
   cnt:=0;ok:=False;
 End;
```

Поясним на примере идею решета. Пусть длина последовательности равна двум, а количество цветов — четырем. Ребенок загадал 32, а компьютер спросил 24. Ответ ребенка 1 0, фиксируем его в переменных *kr* (1) и *bk*(0).

Таблица 3.1

A	*B*	*B* (после анализа *bk*)	*B* (после анализа *kr*)	Результат
11	*True*		*False*	*False*
12	*True*			***True***
13	*True*		*False*	*False*
14	*True*	*False*		*False*
21	*True*	*False*		*False*
22	*True*	*False*		*False*
23	*True*	*False*		*False*
24	*True*	*False*		*False*
31	*True*		*False*	*False*
32	*True*			***True***
33	*True*		*False*	*False*
34	*True*	*False*		*False*
41	*True*			***True***
42	*True*		*False*	*False*
43	*True*			***True***
44	*True*	*False*		*False*

Итак, было 16 возможных вариантов, после первого осталось четыре (см. таблицу 3.1) — работа решета. Функция, реализующая анализ элемента массива *A* по значениям переменных *kr* и *bk*, имеет вид:

```
Function Pr(a,b:Post;kr,bk:Integer):Boolean;
 Var i,x:Integer;
 Begin
{*Проверка по "быкам".*}
  x:=0;
  For i:=1 To 4 Do
    If a[i]=b[i] Then Inc(x);
  If x<>bk Then
    Begin Pr:=False;Exit;End;
     {*Проверка по "коровам".*}
  x:=0;
  For i:=1 To 4 Do
    If (a[i]<>b[i]) And (Pos(b[i],a)<>0)
      Then Inc(x);
  If x<>kr Then
    Begin Pr:=False;Exit;End;
  Pr:=True;
 End;
```

Логика — «сделать отсечение» по значению хода *h*.

```
Procedure Hod(h:Post);
 Var i,kr,bk:Integer;
 Begin
  Inc(cnt);Write(cnt:2,'.',h,'-');
  ReadLn(kr,bk);
  If bk=4 Then
   Begin ok:=True;<вывод результата>;End
         Else
   For i:=1 To Pmax Do
     If B[i] And Not Pr(A[i], h,kr,bk)
       Then B[i]:=False;
 End;
```

Основная программа.

```
.....
Begin
 ClrScr;
 Init;
```

```
Hod('1223');
While Not ok Do
Begin
      выбор очередного хода из A
  Hod(h);
End;
End.
```

Дальнейшая работа с задачей требует ответа на следующие вопросы:

- Зависит ли значение *cnt* от выбора первого хода? Установите экспериментальным путем.
- Как алгоритм выбора очередного хода влияет на результат игры? Исследуйте. Например, выберите тот элемент *A* (соответствующий элемент *B* должен быть равен *True*), в котором количество несовпадающих цифр максимально.

Примечание

Рассмотренные задачи носят учебный характер. Обычно идеи метода решета используются в совокупности с другими методами.

3.7. Задачи

1. Задача о парламенте[*].

На острове Новой Демократии каждый из жителей организовал партию, которую сам и возглавил. Любой из жителей острова может состоять не только в своей партии, но и в других партиях. К всеобщему удивлению, даже в самой малочисленной партии оказалось не менее двух человек. К сожалению, финансовые трудности не позволили создать парламент, куда вошли бы, как предполагалось по Конституции острова, президенты всех партий. Посовещавшись, островитяне решили, что будет достаточно, если в парламенте будет хотя бы один член каждой партии.

Помогите островитянам организовать такой, как можно более малочисленный парламент, в котором будут представлены все партии.

Исходные данные: каждая партия и ее президент имеют один и тот же порядковый номер от 1 до N ($4 \le N \le 150$). Вам даны списки всех N партий острова Новой Демократии. Выведите предлагаемый вами парламент в виде списка номеров ее членов. Например, для четырех партий результат приведен в таблице 3.2.

---

[*] Межгосударственная олимпиада по информатике 1992 года, автор задачи Ю. Корженевич.

Таблица 3.2

Президенты	Члены партий
1	2,3,4
2	3
3	1,4,2
4	2

Список членов парламента 2 состоит из одного жителя с номером 2.

Указание. Задача относится к классу *NP*-полных задач. Ее решение — полный перебор всех вариантов. Начиная с подмножеств мощности 1, требуется генерировать все возможные подмножества множества жителей (до подмножеств мощности *N Div* 2) и заканчивать процесс в том случае, когда будет найдено решение для подмножеств определенной мощности.

Покажем, что ряд эвристик позволяет сократить перебор для некоторых наборов исходных данных.

Представим информацию о партиях и их членах с помощью следующего зрительного образа — таблицы (см. таблицу 3.3). Номером строки является номер партии, номером столбца — номер жителя.

Таблица 3.3

Номер партии	Номер жителя			
	1	2	3	4
1	1	1	1	1
2	0	1	1	0
3	1	1	1	1
4	0	1	0	0

Тогда задачу можно переформулировать следующим образом. Найти минимальное число столбцов, таких, что множество единиц из них «покрывает» множество всех строк. Очевидно, что для нашего примера это один столбец — второй.

Рассмотрим еще один пример (см. рисунок 3.31). Из первоначальной таблицы можно исключить третий столбец, соответствующий третьему жителю, — множество партий, в которых он состоит, содержится в множестве партий, в которых состоит второй житель. После сокращения таблицы могут появиться строки с одной единицей (партия представлена одним жителем). Таких жителей необходимо включать в парламент. Включаем второго жителя. После этого останется представить в парламенте только четвертую партию. Итак:

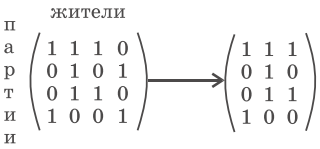


Рис. 3.31

- во-первых, требуется проверять, есть ли житель, представляющий все множество не представленных в парламенте партий;
- во-вторых, исключать часть жителей из рассмотрения, если есть жители, представляющие большее количество партий, и это множество партий содержит партии исключаемого жителя;
- в-третьих, если в результате предыдущей операции появляются партии, представленные одним жителем, то этих жителей обязательно требуется включать в парламент.

Вывод: перебор следует выполнять не по всем жителям и не для всех партий. Второе и третье действия необходимо выполнить до момента начала перебора вариантов (предварительная обработка) и только затем выполнять схему перебора. Первое действие должно выполняться как на стадии предварительной обработки, так и в процессе перебора.

Общая схема перебора реализуется как обычно. Приведем часть описания данных для того, чтобы она была понятна.

```
Const Nmax=150;
Type  Nint=0..Nmax+1;
      Sset=Set Of 0..Nmax;
Var A:Array[Nint] Of <информация по каждому жителю>;
      N:Integer;{*Число жителей на острове*.}
```

Один из вариантов решения.

```
Procedure Solve(k:Nint;Res,Rt:Sset); {*k- номер
элемента массива A; Res - множество партий,
которые представлены в текущем решении; Rt -
множество партий, которые следует "покрыть"
решением; min - минимальное количество членов
в парламенте; mn - число членов парламента
в текущем решении; Rbest - минимальный парламент;
Rwork - текущий парламент; первый вызов -
Solve(1,[],[1..N]).*}
 Var i:Nint;
```

```
Begin
  блок общих отсечений
 If Rt=[]
  Then
   Begin
    If nm<min
     Then
      Begin min:=mn; Rbest:=Rwork End;
   End
   Else
     Begin
       i:=k;
       While i<=N Do
         Begin
           блок отсечений по i
          Include(i);{*Включить жителя, информация
          по которому записана на месте i в
          массиве A, в решение.*}
          Solve(i+1,Res+A[i].part,Rt-A[i].part);
          Exclude(i);{*Исключить из решения.*}
          Inc(i);
         End;
   End;
End;
```

В качестве примера действий в блоке общих отсечений можно взять следующий алгоритм. Подсчитать для каждого значения i ($1 \le i \le t$) множество партий, представляемых жителями, номера которых записаны в элементах массива с i по t (массив *C:Array[1..N] Of Sset*). Тогда, если *Res* — текущее решение, а *Rt* — множество партий, требующих представления, то при $Res+C[i] \le Rt$ решение не может быть получено и эту ветку перебора следует «отсечь».

Пример отсечений по i — если при включении элемента с номером i в решение значение величины *Rt* не изменяется, то это включение жителя, информация по которому записана на месте i в массиве A, бессмысленно.

2. Какое наименьшее число ферзей можно расставить на доске так, чтобы они держали под боем все ее свободные поля? Модификация задачи. Найти расстановку ферзей, которая одновременно решает задачу для досок 9×9, 10×10 и 11×11.

Указание. Задачу можно решать как обычным перебором, так и, представив доску как граф, путем поиска минимального доминирующего множества вершин.

3. Расставить на доске $N \times N (N \le 12)$ N ферзей так, чтобы наибольшее число ее полей оказалось вне боя ферзей.

4. Расставить на доске как можно больше ферзей так, чтобы при снятии любого из них появлялось ровно одно неатакованное поле.

5. **Задача о коне Аттилы** («Трава не растет там, где ступил мой конь!»).

На шахматной доске стоят белый конь и черный король. Некоторые поля доски считаются «горящими». Конь должен дойти до неприятельского короля, повергнуть его и вернуться на исходное место. При этом ему запрещено становиться как на горящие поля, так и на поля, которые уже пройдены.

6. Магараджа — это фигура, которая объединяет в себе ходы коня и ферзя. Для доски 10×10 найти способ расстановки 10 мирных (не бьющих друг друга) магараджей.

7. Пронумеровать позиции в матрице (таблице) размером 5×5 следующим образом. Если номер i $(1 \le i \le 25)$ соответствует позиции с координатами (x,y), вычисляемыми по одному из следующих правил:

$(z,w)=(x\pm3,\ y);$
$(z,w)=(x,\ y\pm3);$
$(z,w)=(x\pm2,\ y\pm2).$

Требуется:

- написать программу, которая последовательно нумерует позиции матрицы 5×5 при заданных координатах позиции, в которой поставлен номер 1 (результаты должны быть представлены в виде заполненной матрицы);
- вычислить число всех возможных расстановок номеров для всех начальных позиций, расположенных в правом верхнем треугольнике матрицы, включая ее главную диагональ.*

---

* Третья международная олимпиада по информатике, 1991 год.

8. **Замок**[*].

На рисунке 3.32 изображен план замка.

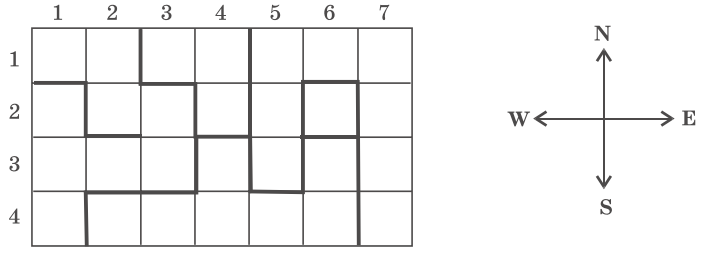


Рис. 3.32

Написать программу, которая определяет:

- количество комнат в замке;
- площадь наибольшей комнаты;
- какую стену в замке следует удалить, чтобы получить комнату наибольшей площади.

Замок условно разделен на $M \times N$ клеток ($M \le 50$, $N \ge 50$). Каждая такая клетка может иметь от 0 до 4 стен.

План замка содержится во входном файле в виде последовательности чисел, по одному числу, характеризующему каждую клетку. В начале файла расположено число клеток в направлении с севера на юг и число клеток в направлении с запада на восток. В последующих строках каждая клетка описывается числом p ($0 \le p \le 15$). Это число является суммой следующих чисел: 1 (если клетка имеет западную стену), 2 (северную), 4 (восточную), 8 (южную). Внутренняя стена считается принадлежащей обеим клеткам. Например, южная стена в клетке (1,1) также является северной стеной в клетке (2,1). Пример входного файла представлен на рисунке 3.33.

Пример.

```
4
7
11  6 11  6  3 10  6
 7  9  6 13  5 15  5
 1 10 12  7 13  7  5
13 11 10  8 10 12 13
```

Рис. 3.33

Замок содержит по крайней мере две комнаты.

В выходном файле должны быть три строки. В первой строке содержится число комнат, во второй — площадь наибольшей комнаты (измеряется количеством клеток). Третья строка содержит три числа, определяющих удаляемую стену: номер

[*] Шестая международная олимпиада по информатике, 1994 год.

строки, номер столбца клетки, содержащей удаляемую стену, и положение этой стены в клетке (*N* — север, *W* — запад, *S* — юг, *E* — восток).

9. **Автозаправка.**

Вдоль кольцевой дороги расположено *M* городов, в каждом из которых есть автозаправочная станция. Известна стоимость *Z*[*i*] заправки в городе с номером *i* и стоимость *C*[*i*] проезда по дороге, соединяющей *i*-й и (*i*+1)-й города, *C*[*M*] — стоимость проезда между первым и *M*-м городами. Для жителей каждого города определить город, в который им необходимо съездить, чтобы заправиться самым дешевым образом, и направление — «по часовой стрелке» или «против часовой стрелки», города пронумерованы по часовой стрелке.

Указание. Переборный вариант решения задачи сводится к проверке всех 2×*M* вариантов для жителей каждого города, итого — 2×*M*×*M* проверок.

Введем два дополнительных массива

```
On, Ag: Array[1..M] Of Record
                        wh, pr:Integer
                        End;.
```

On[*i*] означает, где следует заправляться (*wh*) и стоимость заправки (*pr*) жителям *i*-го города, если движение разрешено только по часовой стрелке. В этом случае жители города *i* имеют две альтернативы: либо заправляться у себя в городе, либо ехать по часовой стрелке. Во втором случае жителям города *i* надо заправляться там же, где и жителям города *i*+1, или в первом, если *i*=*M*. Итак, *On*[*i*]=*min*{*Z*[*i*],*C*[*i*]+*On*[*i*+1].*pr*}. Откуда известно значение *On*[*i*+1].*pr*? Необходимо найти город *j* с минимальной стоимостью заправки — *On*[*j*]:=(*j*,*Z*[*j*]). После этого можно последовательно вычислять значения *On*[*j*−1], *On*[*j*−2], ..., *On*[*j*+1].

Аналогичные действия необходимо выполнить при формировании массива *Ag*[*i*], после этого для жителей каждого города *i* следует выбрать лучший из *On*[*i*].*pr* и *Ag*[*i*].*pr* вариант заправки.

10. В массиве *A* (*Array*[1..*N*,1..*M*] *Of Byte*), заполненном 0 и 1, найти квадратный блок максимального размера, состоящий из одних 0.

Указание. Пусть *N*=5, *M*=6 и массив A заполнен следующим образом (см. таблицу 3.4). Определим массив *B* (*Array*[1..*N*,1..*M*] *Of Byte*) следующим образом. Элементы первой строки и первого столбца инвертируются относительно соответствующих

($B[i,j]=1-A[i,j]$) элементов массива A. И далее, при $i=2..N$ и $j=2..M$ $B[i,j]=0$, если $A[i,j]=1$, и $B[i,j]=Min(B[i-1,j],B[i,j-1],B[i-1,j-1])+1$, если $A[i,j]=0$. Для нашего примера B представлен в таблице 3.5. Ответ задачи — 3.

Таблица 3.4

0	0	0	0	0	1
1	0	0	1	0	0
0	0	0	0	0	1
1	0	0	0	1	0
1	0	0	0	1	1

Таблица 3.5

1	1	1	1	1	0
0	1	2	0	1	1
1	1	2	1	1	0
0	1	2	2	0	1
0	1	2	3	0	0

Как изменится решение, если потребовать нахождение прямоугольного блока максимального размера? Решает ли следующий фрагмент программного кода задачу?

```
Procedure Solve;
  Var i, j, k, nx: Integer;
   Begin
     sqa:=0;{*Результат.*}
     For i:=1 To N Do
     For j:=1 To M Do
       If A[i, j]=0
         Then B[i, j]:=B[i, j-1]+1
         Else
            B[i, j]:=0;{*В массиве A исходные данные,
                 в B - результаты подсчета.*}
            For i:=1 To N Do
            For j:=1 To M Do
              Begin
               nx:=B[i, j];
               For k:=i DownTo 1 Do
                 Begin
                   nx:=Min(nx, B[k, j]);
```

```
        If nx*(i-k+1)>sqa Then sqa:=nx*(i-k+1);
      End;
    End;
  End;
```

11. Задача о паркете*.

Комнату размером $N \times M$ единиц требуется покрыть одинаковыми плитками паркета размером 2×1 единиц без пропусков и наложений ($M \le 20$, $N \le 8$, M, N — целые). Пол можно покрыть паркетом различными способами. Например, для $M=2$, $N=3$ все возможные способы укладки приведены на рисунке 3.34.

Рис. 3.34

Требуется определить количество всех возможных способов укладки паркета для конкретных значений $M \le 20$, $N \le 8$. Результат задачи — таблица, содержащая 20 строк и 8 столбцов.

Элементом таблицы является число, являющееся решением задачи для соответствующих N и M. На месте ненайденных результатов должен стоять символ «*».

Указание. Пусть i — длина комнаты ($1 \le i \le 8$), j — ширина комнаты ($1 \le j \le 20$). «Разрежем» комнату на части. Разрез проводится по вертикали. Плитки, встречающиеся на пути разреза, обходим справа, т. е. при движении сверху вниз на каждом шаге либо обходим справа выступающую клетку, либо нет. При других укладках паркета могут получиться другие сечения. Все варианты сечений легко пронумеровываются, ибо это не что иное, как двоичное число: обход справа плитки соответствует 1, отсутствие обхода — 0. На рисунке 3.35 сечение выделено «жирной» линией, ему соответствует двоичное число 00100011 (35). Количество различных сечений 2^i (пронумеруем их от 0 до 2^i-1), где i — длина комнаты.

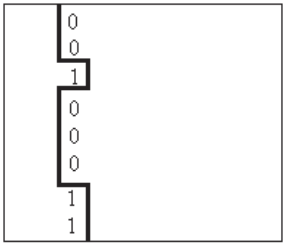


Рис. 3.35

* Задача Сергея Геннадьевича Волченкова с VI Всероссийской олимпиады школьников по информатике 1994 год.

Не будем пока учитывать то, что некоторые из сечений не могут быть получены. Обозначим парой (k,j) комнату с фиксированной длиной i, правый край которой не ровный, а представляет собой сечение с номером k. $B[k,j]$ — количество вариантов укладки паркета в такой комнате. Задача в такой постановке сводится к нахождению $B[0,j]$ — количества укладок паркета в комнате размером (i,j) с ровным правым краем.

Считаем, что $B[0,0]=1$ при любом i, ибо комната нулевой ширины, нулевого сечения и любой длины укладывается паркетом, при этом не используется ни одной плитки. Кроме этого считаем $B[k,0]=0$ для всех сечений с номерами $k<>0$, так как ненулевые сечения при нулевой ширине нельзя реализовать.

Попытаемся найти $B[k,j]$ для фиксированного i. Предположим, что нам известны значения $B[l,j-1]$ для всех сечений с номерами l $(0 \le l \le 2^i-1)$. Сечение l считаем совместимым с сечением k, если путем добавления целого числа плиток паркета из первого можно получить второе. Тогда $B[k,j]=\sum B[l,j-1]$, суммирование ведется по всем сечениям l, совместимым с сечением k. Налицо динамическая схема решения задачи. Схема ее реализации имеет вид:

```
Var     B:Array[0..255,0..20] Of Comp;
        A:Array[1..8,1..20] Of Comp;
          {*Результирующая таблица.*}
Procedure Solve;
 Var i,j,k,l,max:Integer;
 Begin
  For i:=1 To 8 Do
    Begin {*Цикл по значению длины комнаты.*}
      max:=2Shli-1;{*Вычисляем 2 в степени i минус 1. *}
      FillChar(B,SizeOf(B),0);
      B[0,0]:=1;
      For j:=1 To 20 Do
        Begin {*Цикл по значению ширины комнаты.*}
         For k:=0 To max Do {*Сечение с номером k.*}
          For l:=0 To max Do{*Сечение с номером l.*}
           If Can(k,l,i) Then B[k,j]:=B[k,j]+B[l,j-1];
              {*Совместимость сечений "зарыта"
               в функции Can(k,l,i).*}
         A[i,j]:=B[0,j];
        End;
    End;
 End;
```

При решении вопроса о совместимости сечений следует различать понятия совместимость сечений в целом и совместимость в отдельном разряде. При анализе последнего входными данными являются значения разрядов сечений и информация о предыстории процесса (до текущего разряда) — целое или нецелое количество плиток уложено. Выходными данными — признак — целое, нецелое количество плиток, требуемых для перевода сечения l в сечение k, или решение о том, что анализ продолжать не следует — сечения несовместимы. Оставим эту часть работы и доведение схемы решения до работающего варианта на самостоятельное выполнение.

Еще несколько замечаний.

Во-первых, если произведение N на M нечетно (размеры комнаты), то количество укладок паркетом такой комнаты равно 0.

Во-вторых, при M=1 и четном N ответ равен 1.

В-третьих, результирующая таблица симметрична относительно главной диагонали.

В-четвертых, для комнат размером $2\times t$ достаточно просто выводится следующая рекуррентная формула: $A[2,t]=A[2,t-1]+A[2,t-2]$ (ряд Фибоначчи).

Эти замечания реализуются на этапе предварительной обработки и приводят к незначительной модификации алгоритма процедуры *Solve*. Еще одно замечание касается точности результата. Для больших значений N и M необходимо организовать вычисления с использованием «длинной» арифметики (i=8, j=20), ибо результат равен 3547073578562247994.

12. **«Канадские авиалинии»**[*].

Вы победили в соревновании, организованном канадскими авиалиниями. Приз — бесплатное путешествие по Канаде. Путешествие начинается с самого западного города, в который летают самолеты, проходит с запада на восток, пока не достигнет самого восточного города, в который летают самолеты. Затем путешествие продолжается обратно с востока на запад, пока не достигнет начального города. Ни один из городов нельзя посещать более одного раза за исключением начального города, который надо посетить ровно дважды (в начале и в конце путешествия). Вам также нельзя пользоваться авиалиниями других компаний или другими способами передвижения. Задача состоит в следующем: дан список городов и список прямых рейсов между парами городов; найти маршрут, включающий максимальное количество городов и удовлетворяющий вышеназванным условиям.

---

[*] Задача с Пятой Международной олимпиады школьников по информатике 1993 года.

Указание. Пусть нам необходимо попасть из самого западного города в самый восточный, посетив при этом максимальное количество городов. Связи между городами будем записывать с помощью массива *West*:

```
West[i,j]= { True, есть авиалиния между городами
             с номерами i и j;
           { False, авиалинии нет.
```

Пусть мы каким-то образом решили задачу для всех городов, которые находятся западнее города с номером i, т. е. для городов с номерами 1..(i–1). Что значит решили? У нас есть маршруты для каждого из этих городов, и нам известно, через сколько городов они проходят. Обозначим через Q_i множество городов, находящихся западнее i и связанных с городом i авиалиниями (см. рисунок 3.36). Для этих городов задача решена, т. е. известны значения $d[j]$ ($j \in Q_i$) — количество городов в наилучшем маршруте, заканчивающемся в городе с номером j.

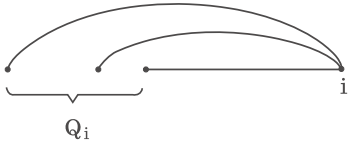


Рис. 3.36

Итак:

```
d[i]:=0;
For j∈Qi Do If d[j]+1>d[i] Then d[i]:=d[j]+1;
```

Остается открытым вопрос, а где же маршрут? Ибо значение d дает только количество городов, через которые он проходит. А для этого необходимо запомнить не только значение *d[i]*, но и номер города j, дающего это значение. Возможно использование следующей структуры данных:

```
A:Array[1..max] Of Record
                  d, l:Byte;
                  End;.
```

В данной схеме мы видим не что иное, как метод динамического программирования в действии. Как обобщить этот набросок решения для первоначальной формулировки задачи? Для удобства перенумеруем города в обратном порядке — с востока на запад. Изменим массив A:

```
Array[1..max,1..max] Of Record
                       d,l:byte;
                       End;
```

Под элементом *A*[*i*,*j*].*d* (путь из города *i* в город *j*) понимается максимальное число городов в маршруте, состоящем из двух частей: от *i* до 1 (на восток) и от 1 до *j* (на запад). По условию задачи нам необходимо найти наилучший по числу городов путь из *N*-го города в *N*-й. Считаем, что *A*[1,1].*d*=1 *и A*[*i*,*j*]=*A*[*j*,*i*]. Через Q_i обозначим множество городов, из которых можно попасть в город *i*. Верны следующие соотношения:

```
A[i,j].d=max(A[k,j].d+1), при k∈Qi,  k<>j, если j<>1;
A[i,i].d=max(A[k,i].d), при i>1 и k∈Qi.
```

13. Решить предыдущую задачу методом полного перебора вариантов.

Указание. Определим структуры данных, а именно:

```
New:Array[1..max] Of Boolean;
way, way_r:Array[1..2*max] Of Byte;
```

Первый массив необходим для хранения признака — посещался или нет город (*New[i]=True* — города с номером *i* нет в маршруте). Во втором и третьем массивах запоминаются по принципу стека текущий и наилучший маршруты соответственно. Для работы со стеками необходимы указатели — переменные *yk* и *yk_max*. Алгоритм перебора поясняется рисунком 3.37. Ищем путь из города с номером *i*.

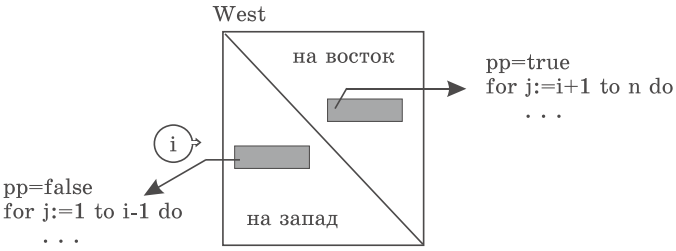


Рис. 3.37

Фрагмент основной части программы.

```
Begin
...
  FillChar(New,SizeOf(New),True);
  yk:=1;Way[yk]:=1;pp:=True;yk_max:=0;
```

```
  Solve(1);
...
End.
```

И процедура поиска решения.

```
Procedure Solve(i:Byte);
 Var j,pn,pv:Byte;
 Begin
  If i=N Then pp:=False; {*Меняем направление.*}
  If (i=1) And Not pp Then
    Begin {*Вернулись в первый город.*}
     If yk>yk_max Then
      Begin {*Запоминаем решение.*}
       yk_max:=yk; way_r:=way;
      End;
      pp:=True;{*Продолжаем перебор.*}
    End;
{*По направлению определяем номера просматриваемых
 городов.*}
  If pp Then
   Begin pn:=i+1; pv:=n; End
        Else
   Begin pn:=1; pv:=i-1; End;
  For j:=pn To pv Do
    If West[i,j] And New[j] Then
       Begin {*В город с номером j летают самолеты
          из города i, и город j еще не посещался.*}
        Inc(yk);Way[yk]:=j;New[j]:=False; {*Включаем
                                  город j в маршрут.*}
        Solve(j);
        New[j]:=True;Way[yk]:=0;Dec(yk); {*Исключаем
                                город j из маршрута.*}
        If j=N Then pp:=True; {*Если исключается
            самый восточный город (N), то меняем
            направление движения.*}
       End;
End;
```

14. Ресторан.

В ресторане собираются N посетителей. Посетитель с номером i подходит во время T_i и имеет благосостояние P_i. Входная дверь ресторана имеет K состояний открытости. Состояние открытости двери может изменяться на одну единицу за одну

единицу времени, т. е. она или открывается на единицу, или закрывается, или остается в том же состоянии. В начальный момент времени дверь закрыта (состояние 0). Посетитель с номером i входит в ресторан только в том случае, если дверь открыта специально для него, то есть когда состояние открытости двери совпадает с его степенью полноты S_i. Если в момент прихода посетителя состояние двери не совпадает с его степенью полноты, то посетитель уходит и больше не возвращается. Ресторан работает в течение времени T.

Цель состоит в том, чтобы, правильно открывая и закрывая дверь, добиться того, чтобы за время работы ресторана в нем собрались посетители с максимальным суммарным благосостоянием.

Входные данные.

В первой строке находятся значения N, K и T, разделенные пробелами ($1 \le N \le 100$, $1 \le K \le 100$, $0 \le T \le 30000$).

Во второй строке находятся времена прихода посетителей T_1, T_2, ...,T_N, разделенные пробелами ($0 \le T_i \le T$, для всех i=1,2,...,N).

В третьей строке находятся величины благосостояния посетителей P_1, P_2, ... P_N, разделенные пробелами ($0 \le P_i \le 300$, для всех i=1,2,...,N).

В четвертой строке находятся значения степени полноты посетителей S_1, S_2, ... S_N, разделенные пробелами ($1 \le S_i \le K$, для всех i=1,2,...,N). Все исходные данные — целые числа.

Выходные данные. Выводится одно число, являющееся максимальным значением суммарного благосостояния посетителей. В случае, если нельзя добиться прихода в ресторан ни одного посетителя, выходной файл должен содержать значение 0.

Классическая задача на метод динамического программирования. Состояния открытости двери можно представить «треугольной решеткой», изображенной на рисунке 3.38. Каждая

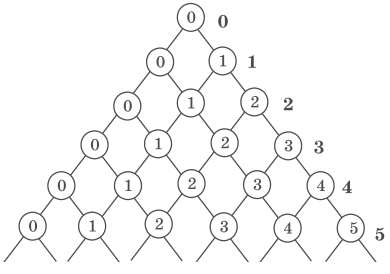


Рис. 3.38

вершина определяет степень открытости q в момент времени t. Некоторым вершинам решетки приписаны веса, равные величине благосостояния посетителя, приходящего в момент времени t и имеющего степень полноты, равную S_i. Что осталось сделать? Найти путь по решетке, проходящий через вершины, сумма весов которых имеет максимальное значение. Следует отметить, что нет необходимости хранить оценки всех вершин. Нам нужно подсчитать оценки для момента времени t. Это возможно, если известны их значения для момента времени $t-1$.

15. Имеется клеточное поле размером $N \times M$. Из каждой клетки можно перемещаться в одну из соседних, если она есть (вверх, вправо, вниз, влево). Коммивояжер стартует из какой-то клетки. Может ли он обойти все клетки и вернуться в исходную клетку?

16. Даны два массива $A[1..N]$ и $B[1..N]$. Матрица расстояний между городами формируется по правилу: $C[i,j]=A[i]+B[j]$ или $C[i,j]=Min(A[i],B[j])$ при $i<>j$. Решить задачу коммивояжера для этих случаев.

17. Черепашка находится в городе, все кварталы которого имеют прямоугольную форму, и ей необходимо попасть с крайнего северо-западного перекрестка на крайний юго-восточный. *Пример*.

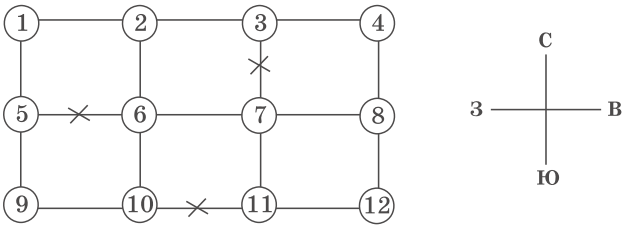


Рис. 3.39

На некоторых улицах проводится ремонт, и по ним запрещено движение (например, на рисунке 3.39 между перекрестками 3 и 7, 5 и 6, 10 и 11), длина, а значит и стоимость проезда по остальным улицам задается. Кроме того, для каждого перекрестка определена стоимость поворота. Так, если Черепашка пришла на 7-й перекресток и поворачивает к 11-му, то она платит штраф, а если идет в направлении 8-го, то платить ей не приходится. Найти для Черепашки маршрут минимальной стоимости.

Исходные данные:

- N — количество перекрестков, определяется через два числа l, m и $N=l\times m$ ($1<l$, $m<11$);
- длина улиц (стоимость проезда) и стоимость поворота на перекрестках — целые числа.

Указание. Определим для каждого перекрестка два числа: стоимость перемещения на юг и стоимость перемещения на восток. Начнем их вычисление с крайнего юго-восточного перекрестка (см. рисунок 3.40). Алгоритм вычислений имеет вид:

```
A[i,j]/восток+ Min(A[i,j+1]/восток,A[i,j+1]/
    юг+A[i,j+1]/поворот)
A[i,j]/юг+Min(A[i+1,j]/юг,A[i+1,j]/восток+A[i+1,
   j]/поворот).
```

При этом i изменяется от $l-1$ до 1, а j от $m-1$ до 1. Иллюстрация к этому алгоритму приведена на рисунке 3.40 (*max* — обозначено отсутствие движения в данном направлении). Итак, стоимость минимального маршрута равна 40. Дальнейшее решение во многом зависит от удачности выбора структур данных. Если определить:

```
Const Nmax=11;
Type Cross=Record
           Right, Down, Turn :Integer
           End;
Var A:Array[1..Nmax,1..Nmax] Of Cross;
```

то программная реализация будет достаточно компактна.

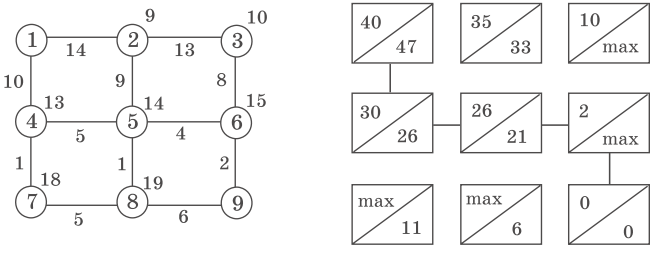


Рис. 3.40

18. Задано прямоугольное клеточное поле $N\times M$ ($2<=N,M<8$) и число k. Построить k различных непрерывных разрезов этого поля на два клеточных поля равной площади. Разрез должен проходить по границам клеток. Пример разреза приведен на рисунке 3.41.

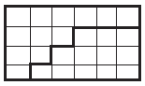


Рис. 3.41

Указание. Задача логически разбивается на следующие части:

- построение разреза между двумя точками на границе клеточного поля;
- подсчет площадей;
- вывод результата.

Наиболее интересной является первая часть. Перенумеруем точки на границе поля по часовой стрелке. После этой операции появляется возможность перебирать по две точки и строить между этими точками разрез. Организация перебора очевидна. Номер первой точки (i) изменяется от 1 до L, где L — количество точек на границе, номер второй от значения i+1 до L. Для построения разреза используется классический волновой алгоритм (задача о лабиринте). Дальнейшее решение зависит от выбора структур данных. Целесообразно хранить координаты точек линии разреза. В этом случае площадь — это сумма площадей прямоугольников.

19. Имеется механизм, состоящий из N расположенных в одной плоскости и свободно надетых на зафиксированные оси пронумерованных шестеренок, зубья которых соприкасаются друг с другом. Информация о механизме ограничивается тем, что для каждой шестеренки перечислены все те, с которыми она находится в непосредственном зацеплении. Напишите программу, определяющую, есть ли такая шестеренка, поворачивая которую мы приведем в движение весь механизм.

20. Дано N конвертов с размерами $A_i \times B_i$ и N прямоугольных открыток с размерами $C_i \times D_i$ i=1, ..., N. Возможно ли разложить открытки по конвертам и если да, то привести все варианты раскладки.

21. **«Американские кубики»**.

Даны четыре кубика — на рисунке 3.42 изображены их развертки. Требуется сложить из них башню 1×1×4 так, чтобы на каждой боковой грани встречались все четыре цифры. Определить форматы входных и выходных данных. Написать программу поиска всех вариантов (если они есть) раскладки кубиков, удовлетворяющих условию задачи.

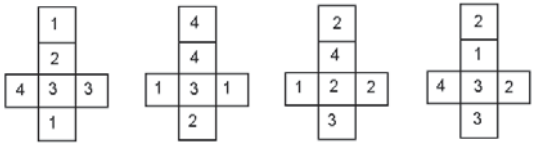


Рис. 3.42

22. Клетки доски 8×8 раскрашены в два цвета: белый и черный. Необходимо пройти из левого нижнего угла в правый верхний так, чтобы цвета клеток перемежались. За один ход разрешается перемещаться на одну клетку по вертикали или горизонтали.

Программа должна (если путь существует):

- находить хотя бы один путь;
- находить путь минимальной длины.

Указание. Для решения задачи применим метод «волны» с учетом условия задачи — чередования клеток разного цвета.

23. Дана матрица $N \times N$ ($N <= 10$), состоящая из целых чисел. За одну операцию разрешается изменить знаки всех чисел произвольно выбранной строки или столбца на противоположные. Требуется сделать суммы чисел во всех строках и столбцах неотрицательными.

Программа должна ввести матрицу и вывести в качестве результата последовательность операций. Количество операций должно быть минимальным.

Указание. Задача решается перебором вариантов. У нас есть $2 \times N$ элементов, N строк и N столбцов. Необходимо генерировать k-элементные подмножества из множества $2 \times N$ элементов, k изменяется от 1 до $2 \times N$, и проверять матрицу после изменения знаков в строках и столбцах, принадлежащих очередному подмножеству.

24. Круг разбили на 8 секторов*. Какие-то два соседних сектора пусты, а в остальных расположены буквы, Р, О, С, С, И, Я в некотором порядке, по одной в секторе (см. рисунок 3.43).

За один ход разрешается взять любые две подряд идущие буквы и перенести их в пустые сектора, сохранив порядок. При этом сектора, которые занимали перенесенные буквы до хода, становятся пустыми. Номером хода назовем номер первого из секторов в порядке обхода по часовой стрелке (ЧС), содержимое которого переносится.

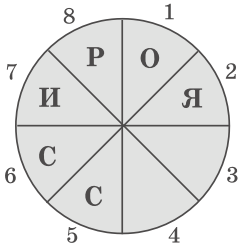


Рис. 3.43

Состояние круга назовем правильным, если начиная с сектора 1 по ЧС можно прочесть слово «РОССИЯ». При этом мес-

---

* Задача предложена для одной из кировских олимпиад по информатике Антоном Валерьевичем Лапуновым.

тоположение пустых секторов может быть произвольным. Состояние круга можно закодировать строкой из 8 символов, получающихся при чтении круга по ЧС, начиная с сектора 1, если пустому сектору поставить в соответствие символ подчеркивания '_'. Эту строку из 8 символов будем называть кодом круга.

По заданному коду найти кратчайшую последовательность ходов, переводящую круг в правильное состояние. Если решений несколько, то необходимо выдать только один вариант.

Указание. Задача решается перебором вариантов. Из каждого состояния круга генерируются все состояния, в которые можно попасть из него. Запоминаем состояния в структуре данных типа очередь. Повторяющиеся состояния в очередь не записываются. Процесс перебора заканчивается при достижении состояния, эквивалентного состоянию «РОССИЯ».

25. Заданы две символьные строки *A* и *B*, не содержащие пробелов. Требуется вычислить, сколькими способами можно получить строку *B* из строки *A*, вычеркивая некоторые символы. Например, если строки *A* и *B* имеют соответственно вид СамаринаИрина и Сара, то искомое число равно 7, для строк *ааавввввссс* и *авс* это число равно 36.

Таблица 3.6

	0	*a*	*a*	*a*	*b*	*b*	*b*	*b*	*c*	*c*	*c*
0	1	1	1	1	1	1	1	1	1	1	1
a	0	1	2	3	3	3	3	3	3	3	3
b	0	0	0	0	3	6	9	12	12	12	12
c	0	0	0	0	0	0	0	0	12	24	36

Написать программу, находящую требуемое число способов.

Указание. Рассмотрим идею решения на примере, приведенном в таблице 3.6. Для клеток, у которых символы на горизонтали и вертикали не совпадают, значение переписывается из соседней по горизонтали клетки. Например, символ '*a*' и первый символ '*b*' по горизонтали. Способов получения строки '*a*' из строки '*aaab*' столько же, сколько из строки '*aaa*'. Для случая совпадения символов, например '*b*' по вертикали и второе '*b*' по горизонтали, количество способов — это сумма способов для строк '*ab*' и '*aaab*' (без последнего символа во второй строке), а также — '*a*' и '*aaab*' (последние символы строк совпадают).

26. В вашем распоряжении имеется по четыре экземпляра каждого из чисел: 2, 3, 4, 6, 7, 8, 9, 10. Требуется написать программу такого размещения их в таблице 6×6 (см. таблицу 3.7), чтобы в клетках, обозначенных одинаковыми значками, находились равные числа и суммы чисел на каждой горизонтали, вертикали и обеих диагоналях совпадали.

Таблица 3.7

+		$	#		11
	@			@	
#	@	11	+		#
		$	11	@	
+	!		#	!	$
11		^	+	^	$

Примечание

Обратите внимание на то, что число 11 уже записано в таблице 4 раза.

Указание. Метод решения задачи традиционный — перебор вариантов, но организовать его следует достаточно грамотно. Первоначально необходимо выбрать числа для помеченных клеток, а затем дополнять до значения суммы элементов, которая вычисляется очень просто — сложить все числа и разделить на шесть (она равна 40).

27. Возьмем клетчатую доску $M \times N$ ($1 \le M,N \le 4$). Воткнем в каждую клеточку штырек. В нашем распоряжении есть K ($0 < K < 20$) абсолютно одинаковых колечек, каждое из которых можно нанизывать на штырек, причем на один штырек разрешается надеть несколько колечек. Подсчитать, сколькими способами можно распределить все эти колечки по штырькам. Два распределения считаются разными, если на каком-то из штырьков находится разное количество колечек, и одинаковыми в противном случае.

Входные данные: *M, N, K*.

Для входных данных: 2, 2, 2, результат: 10.

Указание. Заметим, что клеточная доска дана для «устрашения». С таким же успехом задачу можно решать на линейке из $M \times N$ клеток. Допустим, что найдено решение для L клеток, то есть определено число способов для всех значений колечек от 1 до K. При $L+1$ клетке решение получается очевидным способом — в клетке $L+1$ размещаем p колечек, а в клетках от 1 до L — оставшиеся. Традиционная схема, называемая динамическим программированием.

28. Для известной игры «Heroes of Might and Magic III» генератор случайных карт создает острова, на которых изначально будут расположены герои. Но при случайной генерации карты острова получаются различными по величине. Назовем коэффициентом несправедливости отношение площади наибольшего острова к площади наименьшего. Необходимо подсчитать этот коэффициент. Карта представляет собой прямоугольник $N \times M$, в каждой клетке которого записан 0 (вода) или 1 (земля). Островом считается множество клеток, содержащих 1, таких, что от любой до любой из них можно пройти по клеткам этого множества, переходя только через их стороны.

Входные данные (см. рисунок 3.44). В первой строке входного файла содержатся числа N и M (размеры карты, $1 \le N, M \le 1000$). В следующих M строках записано по N чисел (разделенных пробелами), каждое из которых либо 0, либо 1.

Выходные данные. В выходной файл вывести коэффициент несправедливости с 5 знаками после запятой. Если на карте нет ни одного острова, вывести 0.

Пример.

f.in

```
7 6
1 1 0 0 0 0 0
0 1 0 1 0 0 0
1 1 0 1 1 0 0
1 0 0 0 1 0 0
0 0 0 1 1 1 0
1 1 1 0 0 1 0
```

f.out

```
2.33333
```

Рис. 3.44

Указание. Упростим задачу. Пусть размеры поля не превышают по каждому измерению значения 100. Проблем с оперативной памятью в этом случае нет, и решение хорошо известно — алгоритм волны (выход из лабиринта). Суть: «цепляем» первую ненулевую клетку поля и пытаемся распространять «волну» по всем ненулевым соседним клеткам (обходом в ширину), гася при этом единичные значения в клетках поля и подсчитывая количество таких клеток.

Изменение размерности, а именно $1 \le N, M \le 1000$, приводит к совершенно другому решению. Сложность задачи возрастает на порядок. Описание карты следует считывать по строкам, ибо нет возможности хранить ее полностью в памяти (количество клеток умножим на 4 — по два байта на координату — это приводит к значению 4×10^6, явная нехватка памяти). Что изменится в решении? Пусть данные обработаны из N строк, считываем данные по $(N+1)$-й строке. Каждый остров, сформированный по данным из первых N строк и активный к этому этапу обработки, имеет уникальный идентификатор. Заметим, что если при добавлении строки размер острова не увеличился, то из этого следует, что остров не вышел на $(N+1)$ строку (если это единствен-

ный выход острова на данную строку), его площадь следует проверить на значения минимума, максимума и исключить из дальнейшего рассмотрения.

Как происходит обработка очередной строки? Просматриваем ее слева направо. Если встречаем единичный элемент (остров), то возможны следующие ситуации:

- слева и сверху в предыдущей строке не было единичных элементов — создаем описание нового острова;
- слева или сверху есть единичный элемент (остров) — описание острова существует, присоединяем элемент к острову;
- слева и сверху есть единичные элементы, текущий элемент «склеивает» ранее созданные острова, объединяем их (пример приведен на рисунке 3.45 — обработка «темной» клетки приводит к склеиванию ранее созданных связных областей).

После этих действий необходимо проверить, все ли острова подтвердили свою активность. Если есть острова, не подтвердившие свою активность, то следует проверить их площади на минимум и максимум, откорректировать последние и исключить их из дальнейшей обработки.

29. «Полигон»[*].

Существует игра для одного игрока, которая начинается с задания Полигона с N вершинами. Пример графического представления Полигона при N=4 показан на рисунке 3.46. Для каждой вершины Полигона задается значение — целое число, а для каждого ребра — метка операции + (сложение) либо * (умножение). Ребра Полигона пронумерованы от 1 до N.

Первым ходом в игре удаляется одно из ребер.

Каждый последующий ход состоит из следующих шагов:

- выбирается ребро E и две вершины V_1 и V_2, которые соединены ребром E;

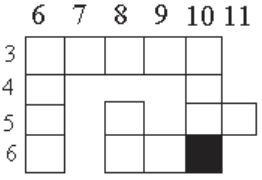


Рис. 3.45

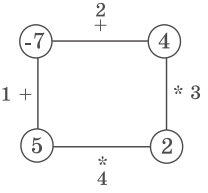


Рис. 3.46

[*] Задача с Международной олимпиады школьников по информатике 1999 года.

- ребро E и вершины V_1 и V_2 заменяются новой вершиной со значением, равным результату выполнения операции, определенной меткой ребра E, над значениями вершин V_1 и V_2.

Игра заканчивается, когда больше нет ни одного ребра. Результат игры — это число, равное значению оставшейся вершины.

Пример игры (см. рисунок 3.47). Рассмотрим полигон, приведенный в условии задачи. Игрок начал игру с удаления ребра 3. После этого игрок выбирает ребро 1, затем — ребро 4 и, наконец, ребро 2. Результатом игры будет число 0.

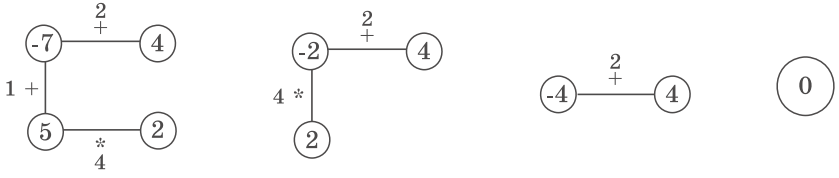


Рис. 3.47

Требуется написать программу, которая по заданному Полигону вычисляет максимальное значение оставшейся вершины и выводит список всех ребер, удаление которых на первом ходе игры позволяет получить это значение.

Входные данные. Файл *pol.in* (см. рисунок 3.48) описывает Полигон с N вершинами. Файл содержит две строки. В первой строке записано число N. Вторая строка содержит метки ребер с номерами 1, ..., N, между которыми записаны через один пробел значения вершин (первое из них соответствует вершине, смежной ребрам 1 и 2, следующее — смежной ребрам 2 и 3 и так далее, последнее — смежной ребрам N и 1). Метка ребра — это буква t, соответствующая операции +, или буква x, соответствующая операции *.

Выходные данные. В первой строке выходного файла *pol.out* (см. рисунок 3.48) программа должна записать максимальное значение оставшейся вершины, которое может быть достигнуто для заданного Полигона. Во второй строке должен быть записан список всех ребер, удаление которых на первом ходе позволяет получить это значение. Номера ребер должны быть записаны в возрастающем порядке и отделяться друг от друга пробелом.

Пример.

pol.in
4
t−7*t*4*x*2*x*5

pol.out
33
1 2

Рис. 3.48

Ограничения. $3 \le N \le 50$. Для любой последовательности ходов значения вершин находятся в пределах [−32768, 32767].

Указание. После удаления первого ребра остается выражение, в котором необходимо так расставить скобки, чтобы его значение было максимальным. Например, пусть удалено первое ребро в примере из формулировки задачи. Остается выражение −7+4×2×5. Способы расстановки скобок: (−7)+(4×2×5), (−7+4)×(2×5), (−7+4×2)×5.

Введем следующий массив

```
Max:Array[1..MaxN,1..MaxN] Of LongInt,
```

где константа *MaxN* равна 50 (по условию задачи). Элемент *Max*[*i*,*j*] определяет максимальное значение выражения длины *j*, *начиная с вершины i* (ключевой момент). Например, *Max*[1,3] — для выражения −7+4×2. Очевидно, что *Max*[*i*,1] равно значениям вершин для всех *i*. Явно просматривается традиционная динамическая схема. Единственная сложность — отрицательные числа. При умножении отрицательных чисел может быть получен максимум. В силу этого необходимо считать и минимальные значения для каждого выражения. Просчитаем наш пример «вручную». Рассмотрим только массив *Max*. Результаты ручного просчета приведены в таблице 3.8. Выбор максимального значения из последнего столбца таблицы дает ответ задачи — максимум выражения и номера ребер.

Таблица 3.8

i \ *j*	1	2	3	4
1	−7	−7+4=-3	−7+(4*2)=1 (−7+4)*2=−6	(−7)+(4*2*5)=33 (−7+4)*(2*5)=−30 (−7+4*2)*(5)=5
2	4	4*2=8	4*(2*5)=40 (4*2)*5=40	(4)*(2*5+(−7))=12 (4*2)*(5+(−7))=−16 (4*2*5)+(−7)=33
3	2	2*5=10	2*(5+(−7))=-4 (2*5)+(−7)=3	(2)*(5+(−7)+4)=4 (2*5)+((−7)+4)=7 (2*5+(−7))+4=7
4	5	5+(−7)=−2	(5)+((−7)+4)=2 (5+(−7))+4=2	(5)+((−7)+4*2)=6 (5+(−7))+(4*2)=6 (5+(−7)+4)*2=4

30. **«Почтовые отделения»**[*].

Вдоль прямой дороги расположены деревни. Дорога представляется целочисленной осью, а расположение каждой деревни задается одним целым числом — координатой на этой оси. Никакие две деревни не имеют одинаковых координат. Расстояние между двумя деревнями вычисляется как модуль разности из координат.

В некоторых, не обязательно во всех, деревнях будут построены почтовые отделения. Деревня и расположенное в ней почтовое отделение имеют одинаковые координаты. Почтовые отделения необходимо расположить в деревнях таким образом, чтобы общая сумма расстояний от каждой деревни до ближайшего к ней почтового отделения была минимальной.

Напишите программу, которая по заданным координатам деревень и количеству почтовых отделений находит такое расположение почтовых отделений по деревням, при котором общая сумма расстояний от каждой деревни до ее ближайшего почтового отделения будет минимальной.

Входные данные. В первой строке файла *post.in* (см. рисунок 3.49) содержатся два целых числа: первое число — количество деревень V, $1 \le V \le 300$, второе число — количество почтовых отделений P, $1 \le P \le 30$, $P \le V$. Вторая строка содержит V целых чисел в возрастающем порядке, являющихся координатами деревень. Для каждой координаты X выполняется $1 \le X \le 10000$.

Пример.

post.in

```
10 5
1 2 3 6 7 9 11 22 44 50
```

post.out

```
9
2 7 22 44 50
```

Рис. 3.49

Выходные данные. Первая строка файла *post.out* (см. рисунок 3.49) содержит одно целое число S — общую сумму расстояний от каждой деревни до ее ближайшего почтового отделения. Вторая строка содержит P целых чисел в возрастающем порядке. Эти числа являются искомыми координатами почтовых отделений. Если для заданного расположения деревень есть несколько решений, то программа должна найти одно из них.

Указание. При размещении одного почтового отделения эта задача известна как задача Лео Мозера (1952 год). Ответ V *Div* 2+1 (V — нечетное число) и любая из двух средних деревень (V — четное). Решение задачи с одним почтовым отделе-

[*] Задача с Международной олимпиады школьников по информатике 2000 года.

нием на взвешенном графе основано на использовании алгоритма Флойда и поиска вершины с минимальной суммой расстояний до других вершин. При увеличении количества почтовых отделений эти идеи «не проходят». Переборные схемы также не работают при данных ограничениях. Остается поиск динамической схемы. Рассмотрим пример, приведенный в формулировке задачи. Размещаем одно почтовое отделение в деревне с номером i. Оценим расстояние до деревень, находящихся левее ее. Результаты оценки приведены в таблице 3.9.

Таблица 3.9

1	2	3	4	5	6	7	8	9	10
0	1	3	12	16	26	38	115	291	345

Что это дает? Пока ничего. Попытаемся решить задачу для двух почтовых отделений. Рассмотрим деревни с номерами от 1 до i. Одно почтовое отделение размещается в деревне с номером i. Найдем место второго почтового отделения, дающего минимальную сумму расстояний. Пример (i равно 5). Крестиками на рисунке 3.50 обозначены деревни с почтовыми отделениями. Справа на рисунке приведены суммы расстояний до ближайших почтовых отделений. Из четырех вариантов выбираем второй, дающий минимальную оценку — 3. Просчитаем для всех значений i. Массив оценок имеет вид, приведенный в таблице 3.10.

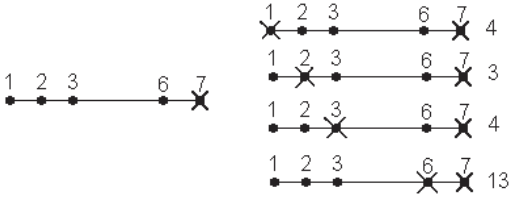


Рис. 3.50

Таблица 3.10

1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	7	12	21	37	43

Символом ∞ обозначен бессмысленный случай — два почтовых отделения в одной деревне.

Продолжим рассмотрение — три почтовых отделения. На рисунке 3.51 приведен случай при i=7 (в 7 деревнях размещаются 3 почтовых отделения). Первый справа отрезок характеризует вариант, при котором два почтовых отделения расположе-

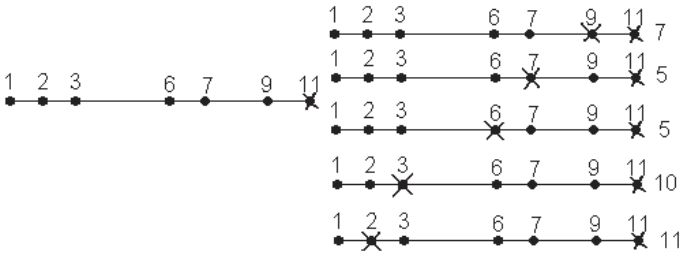


Рис. 3.51

ны в деревнях с 1-й по 6-ю. Он подсчитан (равен 7, третье отделение в седьмой деревне, итоговая оценка не изменяется и равна 7), мы почти «нащупали» динамическую схему. Второй отрезок — два почтовых отделения в деревнях с 1-й по 5-ю. Из предыдущего рассмотрения стоимость этого случая равна 3, расстояние от 6 деревни до ближайшего почтового отделения 2, итого 5. Окончательные оценки при 3, 4 и 5 отделениях приведены в таблице 3.11.

Таблица 3.11

1	2	3	4	5	6	7	8	9	10
∞	∞	0	1	2	3	5	9	21	27
∞	∞	∞	0	1	2	3	5	9	15
∞	∞	∞	∞	0	1	2	3	5	9

Осталось сделать последний шаг. Оценка расстояний в нашей схеме проводилась до деревень, расположенных слева. К этим оценкам следует прибавить сумму расстояний до деревень, находящихся справа. Окончательные оценки приведены в таблице 3.12.

Таблица 3.12

1	2	3	4	5	6	7	8	9	10
∞	∞	∞	∞	101	92	85	53	11	9

Итак, общая сумма расстояний от каждой деревни до ее ближайшего почтового отделения равна 9. Это минимум в окончательном массиве оценок. Как зафиксировать в этой схеме деревни с почтовыми отделениями? Как обычно в алгоритмах типа Флойда, при улучшении оценки, запоминать соответствующую ситуацию.

31. **Палиндром*** — это симметричная строка, т. е. она одинаково читается как слева направо, так и справа налево. Вы должны написать программу, которая по заданной строке определяет минимальное количество символов, которые необходимо вставить в строку для образования палиндрома.

Например, вставкой двух символов строка «*Ab3bd*» может быть преобразована в палиндром («*dAb3bAd*» или «*Adb3bdA*»), а вставкой менее двух символов палиндром в этом примере получить нельзя.

Входные данные. Файл *palin.in* (см. рисунок 3.52) состоит из двух строк. Первая строка содержит одно целое число — длину N входной строки, $3 \le N \le 5000$. Вторая — строку длины N, которая состоит из прописных (заглавных) букв от '*A*' до '*Z*', строчных букв от '*a*' до '*z*' и цифр от '0' до '9'. Прописные и строчные буквы считаются различными.

Пример.

palin.in

5
Ab3bd

palin.out

2

Рис. 3.52

Выходные данные. Файл *palin.out* (см. рисунок 3.52) состоит из одной строки. Эта строка содержит одно целое число, которое является искомым минимальным числом символов.

Указание. Рассмотрим идею решения задачи. Формируем матрицу $R[i,j]$ (i,j=1..N), где $R[i,j]$ — минимальное количество букв, которые необходимо вставить в строку $S(i,j)$ с i-го символа по j-й символ строки S для того, чтобы получить из нее палиндром. Искомый результат — $R[1,N]$.

Как формируется матрица R? Во-первых, заполняется только одна часть матрицы, выше или ниже главной диагонали, включая ее. Во-вторых, если $S[i]=S[j]$, то $R[i,j]=R[i+1,j-1]$. В-третьих, если $S[i]<>S[j]$, то $R[i,j]=Min(R[i+1,j],R[i,j-1])+1$.

Потренируемся на примерах.

S='Ab3bd'. Матрица R имеет вид, приведенный в таблице 3.13.

Таблица 3.13

0	1	2	1	2
-	0	1	0	1
-	-	0	1	2
-	-	-	0	1
-	-	-	-	0

* Задача с Международной олимпиады школьников по информатике 2000 года.

S='abcdba3'. Матрица *R* имеет вид, приведенный в таблице 3.14.

Таблица 3.14

0	1	2	3	2	1	2
-	0	1	2	1	2	3
-	-	0	1	2	3	4
-	-	-	0	1	2	3
-	-	-	-	0	1	2
-	-	-	-	-	0	1
-	-	-	-	-	-	0

Естественно, что хранить в памяти компьютера всю матрицу не следует. Достаточно одномерных массивов.

32. Двое играют в следующую игру: они разложили однокопеечные монетки в стопки (в разных стопках может быть различное количество монет), а стопки расположили в ряд слева направо на столе перед собой. Затем игроки по очереди делают ходы. На каждом ходе участник берет слева несколько стопок (не меньше одной), но не больше, чем перед этим взял его противник (первый игрок своим первым ходом берет не более *K* стопок). Игра заканчивается, когда стопок не осталось. Требуется вычислить, какое максимальное число монет может заполучить первый игрок, если второй тоже старается ходить так, чтобы получить как можно больше.

Входные данные. Во входном файле *input.txt* (см. рисунок 3.53) записано число стопок *N* ($1 \le N \le 180$). Затем *N* чисел, задающих количество монет в стопках слева направо (количество монет в стопке — не менее 1 и не более 20000), и в конце число *K*, ограничивающее количество стопок, которые допускается брать на первом ходе ($1 \le K \le 80$).

Выходные данные. В выходной файл *output.txt* (см. рисунок 3.53) выводится одно число — максимальное количество монет, которое заведомо может заполучить первый игрок, как бы ни играл второй.

Указание. Суть решения отражена в следующей процедуре и выводе результата *A*[1,*K*], где *A — Array*[1..*MaxN*+1, 0..*MaxK*] *Of LongInt*.

Примеры.

input.txt

3 4 9 1 8

output.txt

14

input.txt

4 1 2 2 7 3

output.txt

5

input.txt

5 3 4 8 1 7 2

output.txt

Рис. 3.53

Выполните трассировку процедуры и разработайте программу решения задачи.

```
Procedure Solve;
  Var i, j: Integer;
      sum: LongInt;
   Begin
     sum:=0;
     For i:=N DownTo 1 Do
       Begin
        sum:=sum + Pr[i];
        For j:=1 To K Do
         Begin
          A[i, j]:=A[i, j-1];
          If i+j<N+2 Then
           A[i, j]:=Max(A[i, j], sum - A[i+j, j]);
                    {*Обычная функция определения
                     максимального из двух чисел.*}
         End;
       End;
   End;
```

33. Задача о магических квадратах.

В квадратной таблице $N \times N$ записать числа 1, 2, 3, ..., $N \times N$ так, чтобы суммы по всем столбцам, строкам и главным диагоналям были одинаковы. Примеры магических квадратов приведены на рисунке 3.54.

2	7	6
9	5	1
4	3	8

8	1	6
3	5	7
4	9	2

1	2	15	16
12	14	3	5
13	7	10	4
8	11	6	9

Рис. 3.54

Указание. Задача имеет решение лишь для $N \geq 3$. Сумма всех чисел квадрата равна $N^2 \times (N^2+1)/2$. Таким образом, искомая сумма равна $N \times (N^2+1)/2$. Чудовищный алгоритм решения: вы-

тягиваем квадрат в линейку, генерируем все перестановки чисел 1, 2, 3, ..., $N \times N$ и проверяем каждую перестановку на предмет ее пригодности требованиям магичности квадрата. Не используется даже факт знания значения суммы чисел. Из 9! перестановок при $N=3$ только 8 перестановок удовлетворяют условиям задачи, а из 16!=20922789888000 – 7040 перестановок. Проверьте свой компьютер на выживаемость при $N=4$. Во время его счета оцените вручную временные затраты алгоритма (допускается прибавить к производительности вашего компьютера несколько миллионов операций в секунду). После этого (при условии, что глава вами проработана достаточно тщательно) с помощью ряда эвристик, сокращающих перебор, вы быстро напишете новый вариант алгоритма и получите все решения при $N=4$. О дальнейшем продвижении в сторону увеличения значения N позвольте умолчать.

4. Алгоритмы на графах

Данная глава посвящена алгоритмам на графах, точнее прикладным аспектам этой темы. Если при изложении материала избежать многочисленных дефиниций и доказательств (к чему стремился автор), то он, в силу своей наглядности и простоты, становится доступным и школьнику. Работа по этой проблематике значительно обогащает алгоритмическую культуру учащихся, совершенствует технику написания программ, так как алгоритмы являются неиссякаемым источником задач любого уровня сложности.

4.1. Представление графа в памяти компьютера

Определим *граф* как конечное множество вершин V и набор E неупорядоченных и упорядоченных пар вершин и обозначим $G=(V,E)$.

Количество элементов в множествах V и E будем обозначать буквами N и M.

Неупорядоченная пара вершин называется *ребром*, а упорядоченная пара — *дугой*.

Граф, содержащий только ребра, называется *неориентированным*; граф, содержащий только дуги, — *ориентированным*, или *орграфом*.

Вершины, соединенные ребром, называются смежными. Ребра, имеющие общую вершину, также называются смежными. Ребро и любая из его двух вершин называются *инцидентными*. Говорят, что ребро (u, v) соединяет вершины u и v.

Каждый граф можно представить на плоскости множеством точек, соответствующих вершинам, которые соединены линиями, соответствующими ребрам. В трехмерном пространстве любой граф можно представить таким образом, что линии (ребра) не будут пересекаться.

Способы описания. Выбор соответствующей структуры данных для представления графа имеет принципиальное значение при разработке эффективных алгоритмов. При решении задач используются следующие четыре основных способа описания графа:

- матрица инциденций;

- матрица смежности;
- списки связи;
- перечни ребер.

Мы будем использовать только два: матрицу смежности и перечень ребер.

Матрица смежности — это двумерный массив размерности $N \times N$.

$$A[i,j] = \begin{cases} 1, \text{ вершина с номером } i \text{ смежна с вершиной с номером } j \\ 0, \text{ вершина с номером } i \text{ не смежна с вершиной с номером } j \end{cases}$$

Для хранения перечня ребер необходим двумерный массив R размерности $M \times 2$. Строка массива описывает ребро.

4.2. Поиск в графе

4.2.1. Поиск в глубину

Идея метода. Поиск начинается с некоторой фиксированной вершины v. Рассматривается вершина u, смежная с v. Она выбирается. Процесс повторяется с вершиной u. Если на очередном шаге мы работаем с вершиной q и нет вершин, смежных с q и не рассмотренных ранее (новых), то возвращаемся из вершины q к вершине, которая была до нее. В том случае, когда это вершина v, процесс просмотра закончен.

Очевидно, что для фиксации признака, просмотрена вершина графа или нет, требуется структура данных типа:

```
Nnew : Array[1..N] Of Boolean.
```

Пример. Пусть граф описан матрицей смежности A. Поиск начинается с первой вершины. На рисунке 4.1 приведен исходный граф, а на рисунке 4.2 у вершин в скобках указана та очередность, в которой вершины графа просматривались в процессе поиска в глубину.

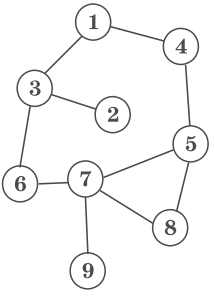


Рис. 4.1

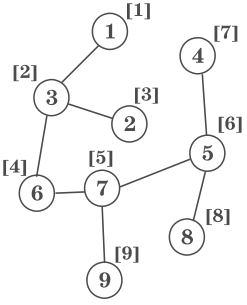


Рис. 4.2

Алгоритм поиска.

```
Procedure Pg(v:Integer);
            {*Массивы Nnew и A глобальные.*}
 Var j:Integer;
 Begin
  Nnew[v]:=False; Write(v:3);
  For j:=1 To N Do
    If (A[v,j]<>0) And Nnew[j]
       Then Pg(j);
 End;
```

Фрагмент основной программы.

```
...
FillChar(Nnew,SizeOf(Nnew),True);
For i:=1 To N Do
  If Nnew[i] Then Pg(i);
...
```

В силу важности данного алгоритма рассмотрим его нерекурсивную реализацию. Глобальные структуры данных прежние: *A* — матрица смежностей; *Nnew* — массив признаков. Номера просмотренных вершин графа запоминаются в стеке *St*, указатель стека — переменная *yk*.

```
Procedure Pgn(v:Integer);
 Var  St:Array[1..N] Of Integer;
      yk, t, j:Integer;
      pp:Boolean;
 Begin
  FillChar(St,SizeOf(St),0); yk:=0;
  Inc(yk);St[yk]:=v;Nnew[v]:=False;
  While yk<>0 Do
   Begin {*Пока стек не пуст.*}
    t:=St[yk];{*Выбор "самой верхней" вершины
              из стека.*}
    j:=1;pp:=False;
    Repeat
     If (A[t,j] <>0) And Nnew[j]
      Then pp:=True
      Else Inc(j);
    Until pp Or (j>=N); {*Найдена новая вершина
           или все вершины, связанные с данной
           вершиной, просмотрены.*}
```

```
    If pp Then
       Begin
        Inc(yk);
        St[yk]:=j;
        Nnew[j]:=False;{*Добавляем номер вершины
                        в стек.*}
       End
          Else Dec(yk); {*"Убираем" номер вершины
                         из стека.*}
  End;
 End;
```

4.2.2. Поиск в ширину

Идея метода. Суть (в сжатой формулировке) заключается в том, чтобы рассмотреть все вершины, связанные с текущей. Принцип выбора следующей вершины — выбирается та, которая была раньше рассмотрена. Для реализации данного принципа необходима структура данных очередь.

Пример. Исходный граф на рисунке 4.3. На рисунке 4.4 рядом с вершинами в скобках указана очередность просмотра вершин графа.

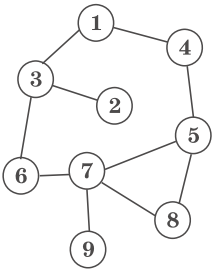


Рис. 4.3

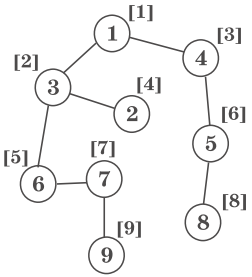


Рис. 4.4

Приведем процедуру реализации данного метода обхода вершин графа.

Алгоритм просмотра вершин.

```
Procedure Pw(v:Integer);
 Var Og:Array[1..N] Of 0..N;{*Очередь.*}
     yk1,yk2:Integer;{*Указатели очереди,
     yk1 - запись; yk2 - чтение.*}
     j:Integer;
 Begin
```

```
  FillChar(Og,SizeOf(Og),0);
  yk1:=0;yk2:=0;{*Начальная инициализация.*}
  Inc(yk1);Og[yk1]:=v;Nnew[v]:=False;
           {*В очередь - вершину v.*}
  While yk2<yk1 Do
    Begin {*Пока очередь не пуста.*}
     Inc(yk2);v:=Og[yk2];Write(v:3);
             {*"Берем" элемент из очереди.*}
     For j:=1 To N Do {*Просмотр всех вершин,
           связанных с вершиной v.*}
      If (A[v,j]<>0) And Nnew[j] Then
        Begin
         {*Если вершина ранее не просмотрена, то
         заносим ее номер в очередь.*}
         Inc(yk1);Og[yk1]:=j;Nnew[j]:=False;
        End;
    End;
End;
```

4.3. Деревья

4.3.1. Основные понятия. Стягивающие деревья

Деревом называют произвольный связный неориентированный граф без циклов. Его можно определить и по-другому: связный граф, содержащий N вершин и $N-1$ ребер. Для произвольного связного неориентированного графа $G=(V,E)$ каждое дерево (V,T), где $T \subseteq E$, называют стягивающим деревом (каркасом, остовом). Ребра такого дерева называют ветвями, а остальные ребра графа — хордами.

Примечание

Понятие цикла будет дано позже. В данной теме ограничимся его интуитивным пониманием.

Число различных каркасов полного связного неориентированного помеченного графа с N вершинами равно N^{N-2}.

Пример. На рисунке 4.5 приведены граф и его каркасы.

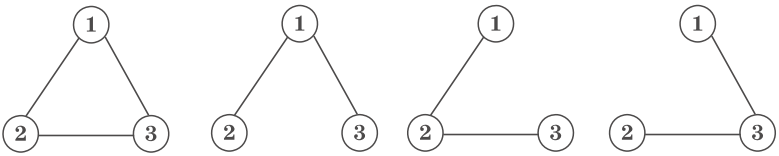


Рис. 4.5

Поиск единственного стягивающего дерева (*каркаса*). Методы просмотра вершин графа поиском в глубину и в ширину позволяют построить одиночный каркас. В вышеописанные программы необходимо вставить запись данных по ребрам, связывающих текущую вершину с ранее не просмотренными в некую структуру данных, например, массив *Tree* (`Array[1..2,1..N] Of Integer`).

Пример. На рисунке 4.6 приведены граф и его каркасы, построенные методами поиска в глубину и в ширину. В круглых скобках указана очередность просмотра вершин графа при соответствующем поиске.

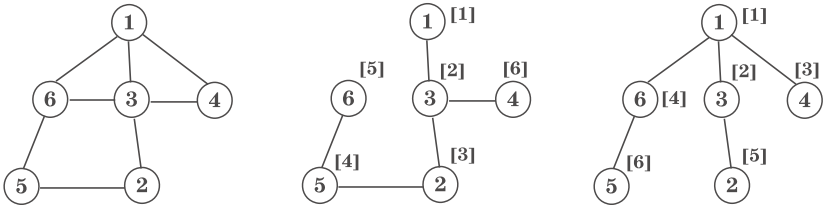


Рис. 4.6

4.3.2. Порождение всех каркасов графа

Дано. Связный неориентированный граф $G=(V,E)$.

Найти. Все каркасы графа.

Каркасы не запоминаются. Их необходимо перечислить. Для порождения очередного каркаса ранее построенные не привлекаются, используется только последний.

Множество всех каркасов графа G делится на два класса: содержащие выделенное ребро (v,u) и не содержащие. Каркасы последовательно строятся в графах $G(v,u)$ и $G-(v,u)$. Каждый из графов $G(v,u)$ и $G-(v,u)$ меньше, чем G. Последовательное применение этого шага уменьшает графы до тех пор, пока не будет построен очередной каркас, либо графы станут несвязными и не имеющими каркасов.

Для реализации идеи построения каркасов графа G используются следующие структуры данных:

- очередь — *Turn (Array[1..N] Of Integer)* с нижним (*Down*) и верхним (*Up*) указателями;
- массив признаков *Nnew*;
- список ребер, образующих каркас, — *Tree*;
- число ребер в строящемся каркасе — *numb*.

Начальное значение переменных:

```
      .....
FillChar(Nnew,SizeOf(Nnew),True);
FillChar(Tree,SizeOf(Tree),0);
Nnew[1]:=False;
Turn[1]:=1; Down:=1;Up:=2; {*В очередь заносим
    первую вершину.*}
numb:=0;
      ......
Procedure Solve(v,q:Integer);{*v - номер вершины,
    из которой выходит ребро. q - номер вершины,
    начиная с которой следует искать очередное
    ребро каркаса .*}
  Var j:Integer;
  Begin
   If Down>=Up Then Exit;
   j:=q;
   While (j<=N) And (numb<N-1) Do
    Begin
    {*Просмотр ребер, выходящих из вершины с номером v.*}
      If (A[v,j]<>0) And Nnew[j] Then
        Begin
        {*Есть ребро, и вершины с номером  j еще нет
        в каркасе. Включаем ребро в каркас.*}
         Nnew[j]:=False;
         Inc(numb);Tree[1,numb]:=v;Tree[2,numb]:=j;
         Turn[Up]:=j;Inc(Up);{*Включаем вершину
                  с номером j в очередь.*}
         Solve(v,j+1); {*Продолжаем построение
                  каркаса.*}
         Dec(Up);Nnew[j]:=True;Dec(numb);
                  {*Исключаем ребро из каркаса.*}
        End;
      Inc(j);
    End;
   If numb=N-1 Then
     Begin <вывод каркаса>; Exit End;
     {*Все ребра, выходящие из вершины с номером v,
      просмотрены. Переходим к следующей вершине
      из очереди и так до тех пор, пока не будет
      построен каркас.*}
   If j=N+1 Then
      Begin
       Inc(Down);
       Solve(Turn[Down],1);
       Dec(Down);
      End;
  End;
```

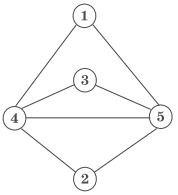


Рис. 4.7

Алгоритм работы процедуры достаточно сложно понимается — двойной рекурсивный вызов. Для этого лучше всего выполнить трассировку работы процедуры на каком-либо примере. Рассмотрим граф, приведенный на рисунке 4.7. Последовательность построения каркасов приведена на рисункх 4.8 и 4.9. Первый каркас строится обычным поиском в ширину. Затем исключаем последнее ребро (4,3) и вновь строим каркас. Первые каркасы, в которых есть ребра (1,4) и (1,5), приведены на следующем рисунке. При исключении ребра (1,5) получаются 8 каркасов.

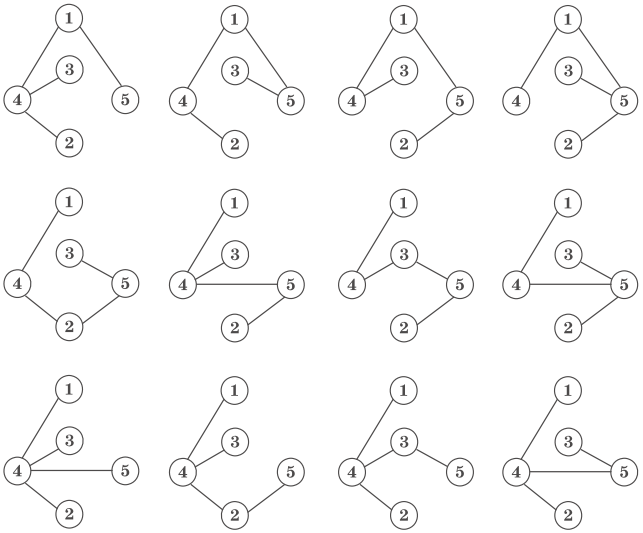


Рис. 4.8

Затем наступает очередь ребра (1,4) и вновь 8 каркасов (см. рисунок 4.9).

4.3.3. Каркас минимального веса. Метод Дж. Краскала

Дано: связный неориентированный граф $G=(V,E)$,
ребра имеют вес,
граф описывается перечнем ребер с указанием их веса,
массив *P (Array[1..3,1..N*(N−1) Div 2] Of Integer)*.

Результат. Каркас с минимальным суммарным весом $Q=(V,T)$, где $T \subset E$.

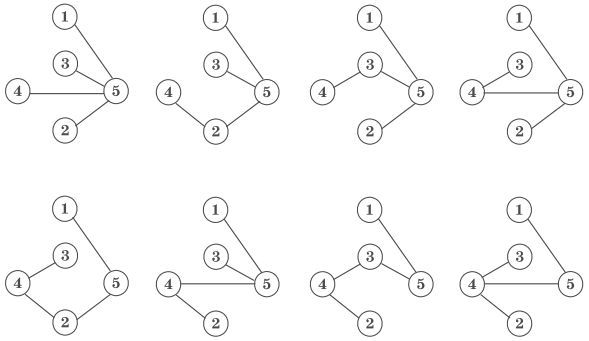


Рис. 4.9

Пример. Граф и процесс построения каркаса по методу Краскала.

Шаг 1. Начать с графа Q, содержащего N вершин и не имеющего ребер.

Шаг 2. Упорядочить ребра графа G в порядке неубывания их весов.

Шаг 3. Начав с первого ребра в этом перечне, добавлять ребра в граф Q, соблюдая условие: добавление не должно приводить к появлению цикла в Q.

Шаг 4. Повторять шаг 3 до тех пор, пока число ребер в Q не станет равным $N-1$. Получившееся дерево является каркасом минимального веса.

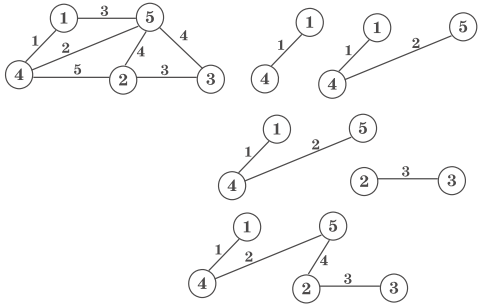


Рис. 4.10

Какие структуры данных требуются для реализации шага 3? Стандартным в программировании методом является введение массива меток вершин графа (*Mark:Array[1..N] Of Integer*). Начальные значения элементов массива равны номерам соответствующих вершин (*Mark[i]=i* для *i* от 1 до *N*). Ребро выбирается в каркас в том случае, если вершины, соединяемые

им, имеют разные значения меток. В этом случае циклы не образуются. Для примера, приведенного на рисунке 4.10, процесс изменения *Mark* показан в таблице 4.1.

Таблица 4.1

Номер итерации	Ребро	Значения элементов *Mark*
Начальное значение	-	[1,2,3,4,5]
1	<1,4>	[1,2,3,1,5]
2	<4,5>	[1,2,3,1,1]
3	<2,3>	[1,2,2,1,1]
4	<2,5>	[1,1,1,1,1]

И алгоритм этого фрагмента.

```
Procedure Chang_Mark(l,m:Integer);
{*Массив Mark глобальный.*}
 Var i,t:Integer;
 Begin
  If m<l Then Begin t:=l;l:=m;m:=t End;
  For i:=1 To N Do
    If Mark[i]=m Then Mark[i]:=l;
 End;
```

Фрагмент основной части программы.

```
Program Tree;
 Const N=..;
 Var   P:Array[1..3,1..N*(N-1) Div 2] Of Integer;
       Mark:Array[1..N] Of Integer;
       k,i,t:Integer;
       M:Integer;{*Количество ребер графа.*}
 Begin
  <ввод описания графа - массив P>;
  <сортировка массива P по значениям весов ребер>;
  For i:=1 To N Do Mark[i]:=i;
  k:=0;t:=M;
  While k<N-1 Do
   Begin
     i:=1;
     While (i<=t) And (Mark[P[1,i]]=Mark[P[2,i]])
        And (P[1,i]<>0) Do Inc(i);
     Inc(k);
     <запоминание ребра каркаса>;
     Change_Mark(Mark[P[1,i]],Mark[P[2,i]]);
   End;
  End;
```

4.3.4. Каркас минимального веса. Метод Р. Прима

Дано: связный неориентированный граф $G=(V,E)$,
ребра имеют вес,
граф описывается матрицей смежности
A (*Array[1..N,1..N] Of Integer*),
элемент матрицы, не равный нулю, определяет вес ребра.

Результат. Каркас с минимальным суммарным весом $Q=(V,T)$, где $T \subset E$.

Отличие от метода Краскала заключается в том, что на каждом шаге строится дерево, а не ациклический граф, т. е. добавляется ребро с минимальным весом, одна вершина которого принадлежит каркасу, а другая нет. Такой принцип «добавления» исключает возможность появления циклов. Для реализации метода необходимы две величины множественного типа *SM* и *SP* (*Set Of 1..N*). Первоначально значением *SM* являются все вершины графа, а *SP* пусто. Если ребро (*i,j*) включается в *T*, то один из номеров вершин *i* и *j* исключается из *SM* и добавляется в *SP* (кроме первого шага, на котором оба номера переносятся в *SP*).

Пример. На рисунке 4.11 приведен граф, для которого последовательно строится каркас. Процесс построения (изменение *SM* и *SP*) показан на рисунках 4.12 и 4.13.

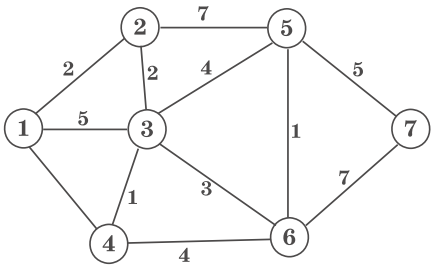


Рис. 4.11

Алгоритм построения каркаса.

```
Procedure Tree; {*A - глобальная структура данных.*}
 Var   SM,SP:Set Of 1..N;
       min,i,j,l,k,t:Integer;
```

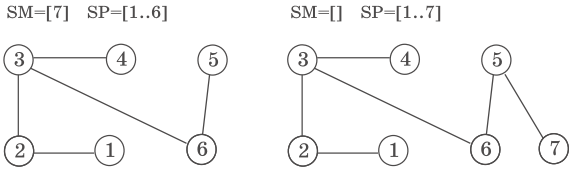


Рис. 4.12

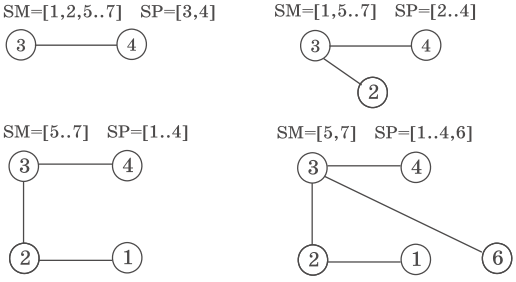


Рис. 4.13

```
Begin
 min:=maxint; SM:=[1..N];SP:=[];{*Включаем первое
                                  ребро в каркас.*}
 For i:=1 To N-1 Do
  For j:=i+1 To N Do
   If (A[i,j]<min) And (A[i,j]<>0) Then
    Begin min:=A[i,j];l:=i;t:=j; End;
 SP:=[l,t];SM:=SM-[l,t]; <выводим или
                          запоминаем ребро <l,t>>;
 {*Основной цикл.*}
 While SM<>[] Do
   Begin
     min:=maxint;l:=0;t:=0;
     For i:=1 To N Do
      If Not(i In SP) Then
        For j:=1 To N Do
         If (j In SP) And (A[i,j]<min) And (A[i,j]<>0)
             Then Begin min:=A[j,k]; l:=i;t:=j;End;
     SP:=SP+[l];SM:=SM-[l];
     {выводим или запоминаем ребро <l,t>};
   End;
End;
```

4.4. Связность

4.4.1. Достижимость

Путем (или ориентированным маршрутом) ориентированного графа называется последовательность дуг, в которой конечная вершина всякой дуги, отличной от последней, является начальной вершиной следующей. Простой путь — это путь, в котором каждая дуга используется не более одного раза. Элементарный путь — это путь, в котором каждая вершина используется не более одного раза. Если существует путь из вершины графа v в вершину u, то говорят, что u достижима из v. Матрицу достижимостей R определим следующим образом:

$$R\,[v,u] = \begin{cases} 1, \text{ если вершина с номером } u \text{ достижима из } v \\ 0, \text{ при недостижимости} \end{cases}$$

Множество $R(v)$ — это множество таких вершин графа G, каждая из которых может быть достигнута из вершины v. Обозначим через $\Gamma(v)$ множество таких вершин графа G, которые достижимы из v с использованием путей длины 1. $\Gamma^2(v)$ — это $\Gamma(\Gamma(v))$, т. е. с использованием путей длины 2 и так далее. В этом случае:

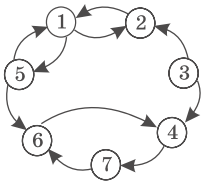


Рис. 4.14

$R(v)=\{v\}\cup\Gamma(v)\cup\Gamma^2(v)\cup...\cup\Gamma^p(v)$.

При этом p — некоторое конечное значение, возможно, достаточно большое.

Пример (см. рисунок 4.14).

$R(1)=\{1\}\cup\{2,5\}\cup\{1,6\}\cup\{2,5,4\}\cup\{1,6,7\}=\{1,2,4,5,6,7\}$.

Выполняя эти действия для каждой вершины графа, мы получаем матрицу достижимостей R.

```
Procedure Reach; {*Формирование матрицы R,
        глобальной переменной. Исходные данные -
        матрица смежности A, глобальная
        переменная.*}
 Var  S,T:Set Of 1..N;
      i,j,l:Integer;
 Begin
  FillChar(R,SizeOf(R),0);
  For i:=1 To N Do
    Begin
    {*Достижимость из вершины с номером i.*}
```

```
    T:=[i];
    Repeat
     S:=T;
     For l:=1 To N Do
      If l In S Then{*По строкам матрицы A,
           принадлежащим множеству S.*}
        For j:=1 To N Do
         If A[l,j]=1 Then T:=T+[j];
    Until S=T;{*Если T не изменилось, то найдены все
          вершины графа, достижимые из вершины
          с номером i.*}
    For j:=1 To N Do
     If j In T Then R[i,j]:=1;
   End;
  End;
```

Матрицу контрдостижимостей Q определим следующим образом:

$$Q[v,u]=\begin{cases}1, \text{ если из } u \text{ можно достигнуть } v\\ 0, \text{ при недостижимости}\end{cases}$$

Множеством $Q(v)$ графа G является множество таких вершин, что из любой его вершины можно достигнуть вершины v. Из определения следует, что столбец v матрицы Q совпадает со строкой v матрицы R, т. е. $Q=R^t$, где R^t — матрица, транспонированная к матрице достижимостей R.

Для примера на рисунке 4.15 приведены матрицы A, R и Q.

$$A=\begin{pmatrix}0&1&0&0&1&0&0\\1&0&0&0&0&0&0\\0&1&0&1&0&0&0\\0&0&0&0&0&0&1\\1&0&0&0&0&1&0\\0&0&0&1&0&0&0\\0&0&0&0&0&1&0\end{pmatrix}\quad R=\begin{pmatrix}1&1&0&1&1&1&1\\1&1&0&1&1&1&1\\1&1&1&1&1&1&1\\0&0&0&1&0&1&1\\1&1&0&1&1&1&1\\0&0&0&1&0&1&1\\0&0&0&1&0&1&1\end{pmatrix}\quad Q=\begin{pmatrix}1&1&1&0&1&0&0\\1&1&1&0&1&0&0\\0&0&1&0&0&0&0\\1&1&1&1&1&1&1\\1&1&1&0&1&0&0\\1&1&1&1&1&1&1\\1&1&1&1&1&1&1\end{pmatrix}$$

Рис. 4.15

Дополнения.

1. Граф называется транзитивным, если из существования дуг (v,u) и (u,t) следует существование дуги (v,t). Транзитивным замыканием графа $G=(V,E)$ является граф $G_z=(V,E\cup E')$, где E' — минимально возможное множество дуг, необходимых для того, чтобы граф G_z был транзитивным.

Задание. Разработать программу для нахождения транзитивного замыкания произвольного графа G.

2. $R(v)$ — множество вершин, достижимых из v, а $Q(u)$ — множество вершин, из которых можно достигнуть u. Определить, что представляет из себя множество $R(v) \cap Q(u)$.

Задание. Разработать программу нахождения этого типа множеств.

4.4.2. Определение связности

Определения.

Неориентированный граф G связен, если существует хотя бы один путь в G между каждой парой вершин i и j.

Ориентированный граф G связен, если неориентированный граф, получающийся из G путем удаления ориентации ребер, является связным.

Ориентированный граф сильно связен, если для каждой пары вершин i и j существует по крайней мере один ориентированный путь из i в j и по крайней мере один из j в i.

Максимальный связный подграф графа G называется связной компонентой графа G. Максимальный сильно связный подграф называется сильно связной компонентой.

Рассмотрим алгоритм нахождения сильно связных компонент графа.

Идея достаточна проста. Для примера на рисунке 4.14 значение $R(1)=\{1\}\cup\{2,5\}\cup\{6\}\cup\{4\}\cup\{7\}=\{1,2,4,5,6,7\}$, а $Q(1)=\{1\}\cup\{2,5\}\cup\{3\}$. Пересечение этих множеств дает множество $C(1)=\{1,2,5\}$ вершин графа G, образующих сильную компоненту, которой принадлежит вершина графа с номером 1. Продолжим рассмотрение: $R(3)=\{1,2,3,4,5,6,7\}$, $Q(3)=\{3\}$ и $C(3)=\{3\}$; $R(4)=\{4\}\cup\{7\}\cup\{6\}=\{4,6,7\}$ и $Q(4)=\{4\}\cup\{6\}\cup\{7\}=\{4,6,7\}$ и $C(4)=\{4,6,7\}$. Итак, мы нашли сильные компоненты графа G.

Граф $G^*=(V^*,E^*)$ определим так: каждая его вершина представляет множество вершин некоторой сильной компоненты графа G, дуга (i^*, j^*) существует в G^* тогда и только тогда, когда в G существует дуга (i,j), такая, что i принадлежит компоненте, соответствующей вершине i^*, а j — компоненте, соответствующей вершине j^*. Граф G^* называется конденсацией графа G.

Некоторые факты по изложенному материалу.

1. G^* не содержит циклов.

2. В ориентированном графе каждая вершина i может принадлежать только одной сильной компоненте.

3. В графе есть множество вершин P, из которых достижима любая вершина графа и которое является минимальным в том смысле, что не существует подмножества в P, обладающего та-

ким свойством достижимости. Это множество вершин называется базой графа.

4. В P нет двух вершин, которые принадлежат одной и той же сильной компоненте графа G. Любые две базы графа G имеют одно и то же число вершин.

Задание. Разработать программу нахождения базы графа.

Схема решения. База P^* конденсации G^* графа G состоит из таких вершин графа G^*, в которые не заходят ребра. Следовательно, базы графа G можно строить так: из каждой сильной компоненты графа G, соответствующей вершине базы P^* конденсации G^*, надо взять по одной вершине — это и будет базой графа G.

4.4.3. Двусвязность

Иногда недостаточно знать, что граф связен. Может возникнуть вопрос, насколько «сильно связен» связный граф. Например, в графе может существовать вершина, удаление которой вместе с инцидентными ей ребрами разъединяет оставшиеся вершины. Такая вершина называется *точкой сочленения*, или *разделяющей вершиной*. Граф, содержащий точку сочленения, называется *разделимым*. Граф без точек сочленения называется двусвязным, или неразделимым. Максимальный двусвязный подграф графа называется двусвязной компонентой, или блоком.

Пример. На рисунке 4.16 показаны разделимый граф и его двусвязные компоненты. Точки сочленения вершины с номерами 4, 5 и 7.

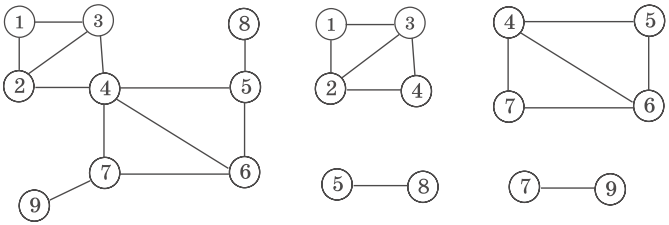


Рис. 4.16

Точку сочленения можно определить иначе. Вершина t является точкой сочленения, если существуют вершины u и v, отличные от t, такие, что каждый путь из u в v (предполагаем, что существует по крайней мере один) проходит через вершину с номером t.

Наша задача — найти точки сочленения и двусвязные компоненты графа.

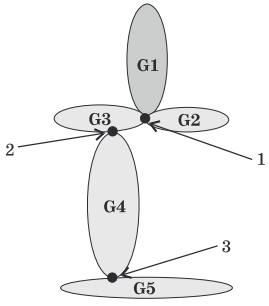


Рис. 4.17

Основная *идея*. Есть двусвязные компоненты G_1, G_2, G_3, G_4 и G_5 и точки сочленения 1, 2, 3 (см. рисунок 4.17). Осуществляем поиск в глубину из вершины t, принадлежащей G_1. Мы можем перейти из G_1 в G_2, проходя через вершину 1. Но по свойству поиска в глубину все ребра G_2 должны быть пройдены до того, как мы вернемся в 1. Поэтому G_2 состоит в точности из ребер, которые проходятся между заходами в вершину 1. Для других чуть сложнее. Из G_1 попадаем в G_3, затем в G_4 и G_5. Предысторию процесса прохождения ребер будем хранить в стеке. Тогда при возвращении в G_4 из G_5 через вершину 3 все ребра G_5 будут на верху стека. После их удаления, т. е. вывода двусвязной компоненты из стека, на верху стека будут ребра G_4, и в момент прохождения вершины 2 мы можем их опять вывести. Таким образом, если распознать точки сочленения, то, применяя поиск в глубину и храня ребра в стеке в той очередности, в какой они проходятся, можно определить двусвязные компоненты. Ребра, находящиеся на верху стека в момент обратного прохода через точку сочленения, образуют двусвязную компоненту.

Итак, проблема с точками сочленения. Рассмотрим граф на рисунке 4.18. В процессе просмотра в глубину все ребра разбиваются на те, которые составляют дерево (каркас), и множество обратных ребер. Обратные ребра выделены на рисунке более жирными линиями.

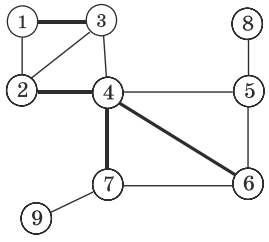


Рис. 4.18

Что нам это дает? Пусть очередность просмотра вершин в процессе поиска в глубину фиксируется метками в массиве *Num*. Для нашего примера *Num* — (1,2,3,4,5,6,7,9,8). Если мы рассматриваем обратное ребро (v,u), и v не предок u, то информацию о том, что $Num[v]$ больше $Num[u]$, можно использовать для пометки вершин v и u как вершин, принадлежащих одной компоненте двусвязности. Массив *Num* использовать для этих целей нельзя, поэтому введем другой массив *Lowpg* и постараемся пометить вершины графа, принадлежащие одной компоненте двусвязности, одним значением метки в этом массиве. Первоначальное значение

метки совпадает со значением соответствующего элемента массива *Num*. При нахождении обратного ребра (*v*,*u*) естественной выглядит операция: *Lowpg*[*v*]:=*Min*(*Lowpg*[*v*],*Num*[*u*]) — изменения значения метки вершины *v*, так как вершины *v* и *u* из одной компоненты двусвязности. К этому алгоритму необходимо добавить смену значения метки у вершины *v* ребра (*v*,*u*) на выходе из просмотра в глубину в том случае, если значение метки вершины *u* меньше, чем метка вершины *v* (*Lowpg*[*v*]:= *Min*(*Lowpg*[*v*],*Lowpg*[*u*])). Для нашего примера массив меток *Lowpg* имеет вид: (1,1,1,2,4,4,4,9,8). Осталось определить момент вывода компоненты двусвязности. Мы рассматриваем ребро (*v*,*u*), и оказывается, что значение *Lowpg*[*u*] больше или равно значению *Num*[*v*]. Это говорит о том, что при просмотре в глубину между вершинами *v* и *u* не было обратных ребер. Вершина *v* — точка сочленения, и необходимо вывести очередную компоненту двусвязности, начинающуюся с вершины *v*.

Итак, алгоритм.

```
Procedure Dvy(v,p:Integer);{*Вершина p - предок
      вершины v. Массивы A, Num, Lowpg и переменная
      nm - глобальные.*}
 Var u:Integer;
 Begin
  Inc(nm);Num[v]:=nm;Lowpg[v]:=Num[v];
  For u:=1 To N Do
   If A[v,u]<>0 Then
    If Num[u]=0
      Then
        Begin
          <сохранить ребро (v,u) в стеке>;
          Dvy(u,v);
          Lowpg[v]:=Min(Lowpg[v],Lowpg[u]); {*Функция,
            определяющая минимальное из двух чисел.*}
          If Lowpg[u]>=Num[v] Then <вывод компоненты>;
        End
      Else
        If (u<>p) And (Num[v]>Num[u]) Then
          Begin
            {*u не совпадает с предком вершины v.*}
              <сохранить ребро (v,u) в стеке>;
             Lowpg[v]:=Min(Lowpg[v],Num[u]);
          End;
 End;
```

Фрагмент основной программы:

```
....
FillChar(Num,SizeOf(Num),0);
FillChar(Lowpg,SizeOf(Lowpg),0);
nm:=0;
For v:=1 To N Do
  If Num[v]=0 Then Dvy(v,0);
....
```

Мостом графа G называется каждое ребро, удаление которого приводит к увеличению числа связных компонент графа.

Задание. Разработать программу нахождения всех мостов графа. Покажите, что мосты графа должны быть в каждом каркасе графа G. Каким образом знание мостов графа может изменить (ускорить) алгоритм нахождения всех его каркасов?

4.5. Циклы

4.5.1. Эйлеровы циклы

Эйлеров цикл — это такой цикл, который проходит ровно один раз по каждому ребру.

Теорема. Связный неориентированный граф G содержит эйлеров цикл тогда и только тогда, когда число вершин нечетной степени равно нулю.

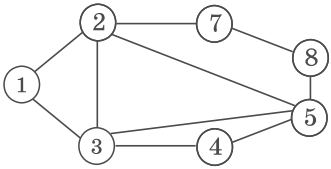
Не все графы имеют эйлеровы циклы, но если эйлеров цикл существует, то это означает, что, следуя вдоль этого цикла, можно нарисовать граф на бумаге, не отрывая от нее карандаша.

Дан граф G, удовлетворяющий условию теоремы. Требуется найти эйлеров цикл. Используется просмотр графа методом поиска в глубину, при этом ребра удаляются. Порядок просмотра (номера вершин) запоминается. При обнаружении вершины, из которой не выходят ребра, мы их удалили, ее номер записывается в стек, и просмотр продолжается от предыдущей вершины. Обнаружение вершин с нулевым числом ребер говорит о том, что найден цикл. Его можно удалить, четность вершин (количество выходящих ребер) при этом не изменится. Процесс продолжается до тех пор, пока есть ребра. В стеке после этого будут записаны номера вершин графа в порядке, соответствующем эйлерову циклу.

```
Procedure Search(v:Integer);{*Глобальные
      переменные: A - матрица смежности, Cv - стек;
      yk - указатель стека.*}
```

```
Var j:Integer;
Begin
 For j:=1 To N Do
  If A[v,j]<>0 Then
    Begin
      A[v,j]:=0;A[j,v]:=0;
      Search(j)
    End;
 Inc(yk);Cv[yk]:=v;
End;
```

На рисунке 4.19 приведен пример графа и содержимое стека *Cv* после работы процедуры *Search*.



Cv: 1 3 5 6 7 2 5 4 3 2 1

Рис. 4.19

4.5.2. Гамильтоновы циклы

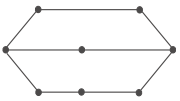


Рис. 4.20

Граф называется *гамильтоновым*, если в нем имеется цикл, содержащий каждую вершину этого графа. Сам цикл также называется гамильтоновым. Не все связные графы гамильтоновы.

На рисунке 4.20 дан пример графа, не имеющего гамильтонова цикла.

Дан связный неориентированный граф *G*. Требуется найти все гамильтоновы циклы графа, если они есть.

Метод решения — перебор с возвратом (*backtracking*). Начинаем поиск решения, например, с первой вершины графа. Предположим, что уже найдены первые k компонент решения. Рассматриваем ребра, выходящие из последней вершины. Если есть такие, что идут в ранее не просмотренные вершины, то включаем эту вершину в решение и помечаем ее как просмотренную. Получена (k+1) компонента решения. Если такой вершины нет, то возвращаемся к предыдущей вершине и пытаемся найти ребро из нее, выходящее в другую вершину. Решение получено при просмотре всех вершин графа и возможности достичь из последней первой вершины. Решение (цикл) выводится, и продолжается процесс нахождения следующих циклов.

На рисунке 4.21 приведен пример графа и часть дерева перебора вариантов, показывающего механизм работы данного метода.

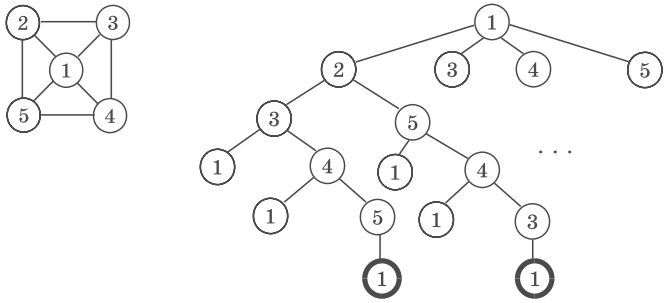


Рис. 4.21

```
Procedure Gm(k:Integer); {* k - номер итерации.
Глобальные переменные: A - матрица смежности; St -
массив для хранения порядка просмотра вершин
графа; Nnew - массив признаков: вершина
просмотрена или нет.*}

 Var j,v:Integer;
 Begin
  v:=St[k-1]; {*Номер последней вершины.*}
  For j:=1 To N Do
   If (A[v,j]<>0) Then {*Есть ребро между
         вершинами с номерами v и j.*}
    If (k=N+1) And (j=1)
     Then <вывод цикла>
     Else
      If Nnew[j] Then
        Begin {*Вершина не просмотрена.*}
          St[k]:=j;
          Nnew[j]:=False;
          Gm(k+1);
          Nnew[j]:=True;
        End;
 End;
```

Фрагмент основной программы.

```
......
St[1]:=1;Nnew[1]:=False; Gm(2);
......
```

4.5.3. Фундаментальное множество циклов

Каркас (V,T) связного неориентированного графа $G=(V,E)$ содержит $N-1$ ребро, где N — количество вершин G. Каждое ребро, не принадлежащее T, т. е. любое ребро из $E-T$, порождает в точности один цикл при добавлении его к T. Такой цикл является элементом фундаментального множества циклов графа G относительно каркаса T. Поскольку каркас состоит из $N-1$ ребра, в фундаментальном множестве циклов графа G относительно любого каркаса имеется $M-N+1$ циклов, где M — количество ребер в G.

Пример графа, его каркаса и множества фундаментальных циклов приведен на рисунке 4.22.

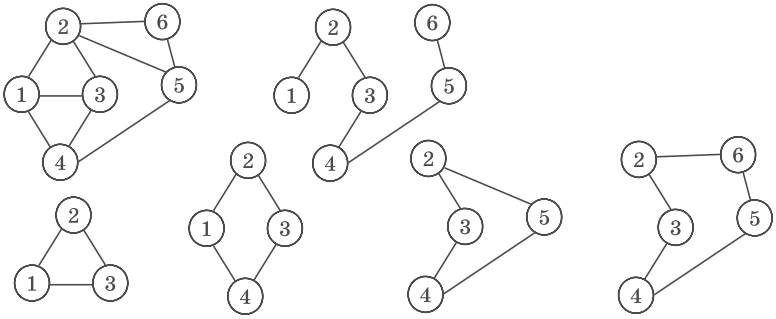


Рис. 4.22

Поиск в глубину является естественным подходом, используемым для нахождения фундаментальных циклов. Строится каркас, а каждое обратное ребро порождает цикл относительно этого каркаса. Для вывода циклов необходимо хранить порядок обхода графа при поиске в глубину (номера вершин) — массив *St*, а для определения обратных ребер вершины следует «метить» (массив *Gnum*) в той очередности, в которой они просматриваются. Если для ребра (v,j) оказывается, что значение метки вершины с номером j меньше, чем значение метки вершины с номером v, то ребро обратное и найден цикл.

Начальная инициализация переменных.

```
...
num:=0;yk:=0;
For j:=1 To N Do Gnum[j]:=0;
...
```

Основной алгоритм.

```
Procedure Circl(v:integer);{*Глобальные
переменные: A - матрица смежности графа; St -
массив для хранения номеров вершин графа в том
порядке, в котором они используются при построении
каркаса; yk - указатель записи в массив St; Gnum -
для каждой вершины в соответствующем элементе
массива фиксируется номер шага (num), на котором
она просматривается при поиске в глубину.*}
 Var j:Integer;
 Begin
  Inc(yk);St[yk]:=v;
  Inc(num);Gnum[v]:=num;
  For j:=1 To N Do
   If A[v,j]<>0 Then
    If Gnum[j]=0
      Then Circl[j] {*Вершина j не просмотрена.*}
       Else
         If (j<>St[yk-1]) And (Gnum[j]<Gnum[v])
           Then <вывод цикла из St>{*j не предыдущая
              вершина при просмотре, и она была
              просмотрена ранее.*};
  Dec(yk);
 End;
```

Название «фундаментальный» связано с тем, что каждый цикл графа может быть получен из циклов этого множества. Для произвольных множеств A и B определим операцию симметрической разности $A \oplus B = (A \cup B) \setminus (A \cap B)$. Известно, что произвольный цикл графа G можно однозначно представить как симметрическую разность некоторого числа фундаментальных циклов. Однако не при всех операциях симметрической разности получаются циклы (вырожденный случай).

Задание. Разработать программу нахождения всех циклов графа.

4.6. Кратчайшие пути

4.6.1. Постановка задачи. Вывод пути

Дано:
ориентированный граф $G=(V,E)$,
веса дуг — $A[i,j]$ ($i,j=1..N$, где N — количество вершин графа),
начальная и конечная вершины — s, $t \in V$.

Веса дуг записаны в матрице смежности A, если вершины i и j не связаны дугой, то $A[i,j]=\infty$. Путь между s и t оценивается $\sum_{i,j \in \text{пути}} A[i,j]$.

Результат: путь с минимальной оценкой.

Пример. Кратчайший путь из 1 в 4 проходит через 3-ю и 2-ю вершины и имеет оценку 6 (см. рисунок 4.23).

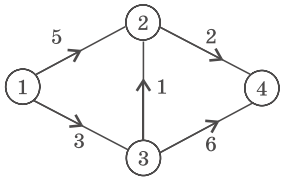


Рис. 4.23

Особый случай — контуры с отрицательной оценкой.

Пример.

При $s=1$ и $t=5$, обходя контур $3\to4\to2\to3$ (см. рисунок 4.24) достаточное число раз, можно сделать так, что оценка пути между вершинами 1 и 5 будет меньше любого целого числа. Оценку пути назовем его весом или длиной. Будем рассматривать только графы без контуров отрицательного веса.

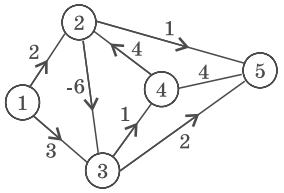


Рис. 4.24

Нам необходимо найти кратчайший путь, т. е. путь с минимальным весом, между двумя вершинами графа. Эта задача разбивается на две подзадачи: сам путь и значение минимального веса.

Обозначим минимальные веса через $D[s,t]$. Неизвестны алгоритмы, определяющие только $D[s,t]$, все они определяют оценки от вершины s до всех остальных вершин графа. Определим D как *Array*[1..*N*] *Of Integer*. Предположим, что мы определили значения элементов массива D — решили вторую подзадачу. Определим сам кратчайший путь. Для s и t существует такая вершина v, что $D[t]=D[v]+A[v,t]$. Запомним v (например, в стеке). Повторим процесс поиска вершины u, такой, что $D[v]=D[u]+A[u,v]$, и так до тех пор, пока не дойдем до вершины с номером s. Последовательность t, v, u, ..., s дает кратчайший путь.

```
Procedure Way(s,t:Integer);{*D, A - глобальные
      структуры данных. St - локальная структура
      данных для хранения номеров вершин.*}
 Var v,u:Integer;
  Procedure Print; {*Выводит содержимое St.*}
   Begin
...
   End;
```

```
Begin
 <почистить St>;
 <занести вершину с номером t в St>;
 v:=t;
 While v<>s Do
   Begin
     u:=<номер вершины, для которой D[v]=D[u]+A[u,v]>;
     <занести вершину с номером v в St>;
     v:=u;
   End;
End;
```

Итак, путь при известном D находить мы умеем. Осталось научиться определять значения кратчайших путей, т. е. элементы массива D. *Идея* всех известных алгоритмов заключается в следующем. По данной матрице весов A вычисляются первоначальные верхние оценки. А затем пытаются их улучшить до тех пор, пока это возможно. Поиск улучшения, например для $D[v]$, заключается в нахождении вершин u, таких, что $D[u]+A[u,v]<D[v]$. Если такая вершина u есть, то значение $D[v]$ можно заменить на $D[u]+A[u,v]$.

4.6.2. Алгоритм Дейкстры

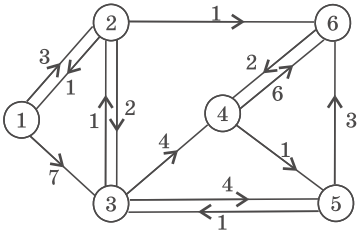
Дано:
ориентированный граф $G=(V,E)$,
s — вершина-источник; матрица смежности A (*Array*[1..N,1..N] *Of Integer*);
для любых $u, v \in V$ вес дуги неотрицательный ($A[u,v] \geq 0$).

Результат: массив кратчайших расстояний D.

В данном алгоритме формируется множество вершин T, для которых еще не вычислена оценка расстояние и, во-вторых (это главное), минимальное значение в D по множеству вершин, принадлежащих T, считается окончательной оценкой для вершины, на которой достигается этот минимум. С точки зрения здравого смысла этот факт достаточно очевиден. Другой «заход» в эту вершину возможен по пути, содержащему большее количество дуг, а так как веса неотрицательны, то и оценка пути будет больше.

Пример. Граф и его матрица смежности.

В таблице 4.2 приведена последовательность шагов (итераций) работы алгоритма. На первом шаге минимальное значение D достигается на второй вершине. Она исключается из множества T, и улучшение оценки до оставшихся вершин (3,4,5,6) ищется не по всем вершинам, а только от второй.



$$A=\begin{pmatrix} \infty & 3 & 7 & \infty & \infty & \infty \\ 1 & \infty & 2 & \infty & \infty & 1 \\ \infty & 1 & \infty & 4 & 4 & \infty \\ \infty & \infty & \infty & \infty & 1 & 5 \\ \infty & \infty & 1 & \infty & \infty & 3 \\ \infty & \infty & \infty & 2 & \infty & \infty \end{pmatrix}$$

Рис. 4.25

Таблица 4.2

№ итерации	*D[1]*	*D[2]*	*D[3]*	*D[4]*	*D[5]*	*D[6]*	*T*
1	0	3	7	∞	∞	∞	[2,3,4,5,6]
2	0	3	5	∞	∞	4	[3,4,5,6]
3	0	3	5	6	∞	4	[3,4,5]
4	0	3	5	6	9	4	[4,5]
5	0	3	5	6	7	4	[5]

```
Procedure Dist;{*A, D, s, N - глобальные величины.*}
 Var  i,u: Integer;
      T:Set Of 1..N;
 Begin
  For i:=1 To N Do D[i]:=A[s,i];
  D[s]:=0;
  T:=[1..N]-[s];
  While T<>[ ] Do
    Begin
      u:=<то значение l, при котором достигается
          min(D[l])>;
      T:=T-[u];
      For i:=1 To N Do
       If i In T Then D[i]:=min(D[i],D[u]+A[u,i]);
    End;
 End;
```

Время работы алгоритма $t \approx O(N^2)$.

4.6.3. Пути в бесконтурном графе

Дано:
ориентированный граф $G=(V,E)$ без контуров,
веса дуг произвольны.

Результат: массив кратчайших расстояний (длин) D от фиксированной вершины s до всех остальных.

Утверждение — в произвольном бесконтурном графе вершины можно перенумеровать так, что для каждой дуги (*i*,*j*) номер вершины *i* будет меньше номера вершины *j*.

Пример.

Введем следующие структуры данных:

- массив *NumIn*, *NumIn*[*i*] определяет число дуг, входящих в вершину с номером *i*;
- массив *Num*, *Num*[*i*] определяет новый номер вершины *i*;
- массив *St*, для хранения номеров вершин, в которые заходит нулевое количество дуг. Работа с массивом осуществляется по принципу стека;
- переменная *nm*, текущий номер вершины.

Идея алгоритма. Вершина *i*, имеющая нулевое значение *NumIn* (а такая вершина на начальном этапе обязательно есть в силу отсутствия контуров в графе), заносится в *St*, ей присваивается текущее значение *nm* (запоминается в *Num*), и изменяются значения элементов *NumIn* для всех вершин, связанных с *i*. Процесс продолжается до тех пор, пока *St* не пуст.

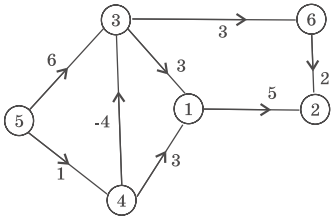


Рис. 4.26

На рисунке 4.26 приведен пример графа, а в таблице 4.3 представлены результаты трассировки работы алгоритма для этого примера.

Таблица 4.3

№ итерации	*NumIn*	*Num*	*St*	*Nm*
начальная	[2,2,2,1,0,1]	[0,0,0,0,0,0]	[5]	0
1	[2,2,1,0,0,1]	[0,0,0,0,1,0]	[4]	1
2	[1,2,0,0,0,1]	[0,0,0,2,1,0]	[3]	2
3	[0,2,0,0,0,0]	[0,0,3,2,1,0]	[6,1]	3
4	[0,1,0,0,0,0]	[0,0,3,2,1,4]	[1]	4
5	[0,0,0,0,0,0]	[5,0,3,2,1,4]	[2]	5
6	[0,0,0,0,0,0]	[5,6,3,2,1,4]	[ ]	6

```
Procedure Change_Num;{*A, Num — глобальные
      структуры данных.*}
 Var NumIn,St:Array[1..N] Of Integer;
     i,j,u,nm,yk:Integer;
 Begin
  FillChar(NumIn,SizeOf(NumIn),0);
  For i:=1 To N Do
   For j:=1 To N Do
    If A[i,j]<>0 Then Inc(NumIn[j]);
    nm:=0;yk:=0;
    For i:=1 To N Do
     If NumIn[i]=0 Then
      Begin Inc(yk);Stack[yk]:=i; End;
    While yk<>0 Do
      Begin
        u:=Stack[yk];Dec[yk];Inc(nm);Num[u]:=nm;
       For i:=1 To N Do
        If A[u,i]<>0 Then
          Begin
           Dec(NumIn[i]);
           If NumIn[i]=0 Then
            Begin Inc(yk); Stack[yk]:=i; End;
          End;
      End;
 End;
```

Итак, пусть для графа G выполнено условие утверждения (вершины перенумерованы) и нам необходимо найти кратчайшие пути (их длины) от первой вершины до всех остальных. Пусть мы находим оценку для вершины с номером i. Достаточно просмотреть вершины, из которых идут дуги в вершину с номером i. Они имеют меньшие номера, и оценки для них уже известны. Остается выбрать меньшую из них.

```
Procedure Dist;{*D, A - глобальные величины.*}
 Var i,j:Integer;
 Begin
  D[1]:=0;
  For i:=2 To N Do D[i]:=MaxInt-<максимальное
        значение в матрице смежности
        A>;{*Определите, с какой целью вычитается
        из MaxInt максимальный элемент матрицы A.*}
  For i:=2 To N Do
   For j:=1 To i-1 Do
```

```
    If A[j,i]<>∞
     Then D[i]:=Min(D[i],D[j]+A[j,i]);
End;
```

Процедура написана в предположении о том, что i и j — новые номера вершин и $A[i,j]$ соответствует этим номерам. Однако это не так. Новые номера по результатам работы предыдущей процедуры хранятся в массиве Num. Требуется «стыковка» новых номеров и матрицы A. Как это сделать наилучшим образом?

Время работы алгоритма $t \approx O(N^2)$.

4.6.4. Кратчайшие пути между всеми парами вершин. Алгоритм Флойда

Дано:

ориентированный граф $G=(V,E)$ с матрицей весов $A(Array[1..N,1..N]\ Of\ Integer)$.

Результат: матрица D кратчайших расстояний между всеми парами вершин графа и кратчайшие пути.

Идея алгоритма. Обозначим через $D^m[i,j]$ оценку кратчайшего пути из i в j с промежуточными вершинами из множества $[1..m]$. Тогда имеем:

$D^0[i,j]:=A[i,j]$ и

$D^{(m+1)}[i,j]= Min\{D^m[i,j],D^m[i,m+1] + D^m[m+1,j]\}$.

Второе равенство требует пояснения. Пусть мы находим кратчайший путь из i в j с промежуточными вершинами из множества $[1..(m+1)]$. Если этот путь не содержит вершину $(m+1)$, то $D^{(m+1)}[i,j]=D^m[i,j]$. Если же он содержит эту вершину, то его можно разделить на две части: от i до $(m+1)$ и от $(m+1)$ до j.

Время работы алгоритма $t \approx O(N^3)$.

```
Procedure Dist; {*A, D - глобальные структуры
      данных.*}
 Var m,i,j:Integer;
 Begin
  For i:=1 To N Do
   For j:=1 To N Do D[i,j]:=A[i,j];
  For i:=1 To N Do D[i,i]:=0;
  For m:=1 To N Do
   For i:=1 To N Do
     For j:=1 To N Do D[i,j]:=Min{D[i,j],
       D[i,m]+D[m,j]};
 End;
```

Пример. На рисунке 4.27 представлены граф и значения матриц типа D при работе процедуры.

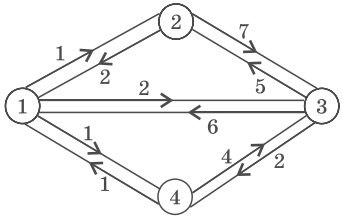


Рис. 4.27

Верхний индекс у D (см. рисунок 4.28) указывает номер итерации (значение m в процедуре *Dist*).

$$D^0=\begin{pmatrix}0&1&2&1\\2&0&7&\infty\\6&5&0&2\\1&\infty&4&0\end{pmatrix}\quad D^1=\begin{pmatrix}0&1&2&1\\2&0&4&3\\6&5&0&2\\1&2&3&0\end{pmatrix}\quad D^2=D^1\quad D^3=D^2\quad D^4=\begin{pmatrix}0&1&2&1\\2&0&4&3\\3&4&0&2\\1&2&3&0\end{pmatrix}$$

Рис. 4.28

Расстояния между парами вершин дает D. Для вывода самих кратчайших путей введем матрицу M того же типа, что и D. Элемент $M[i,j]$ определяет предпоследнюю вершину кратчайшего пути из i в j.

Процедура *Dist* претерпит небольшие изменения. В том случае, когда $D[i,j]$ больше $D[i,m]+D[m,j]$, изменяется не только $D[i,j]$, но и $M[i,j]$. $M[i,j]$ присваивается значение $M[m,j]$. Для нашего примера изменения M показаны на рисунке 4.29.

$$M^0=\begin{pmatrix}1&1&1&1\\2&2&2&2\\3&3&3&3\\4&4&4&4\end{pmatrix}\quad M^1=\begin{pmatrix}1&1&1&1\\2&2&1&1\\3&3&3&3\\4&1&1&4\end{pmatrix}\quad M^2=M^1\quad M^3=M^2\quad M^4=\begin{pmatrix}1&1&1&1\\2&2&1&1\\4&1&3&3\\4&1&1&4\end{pmatrix}$$

Рис. 4.29

Например, необходимо вывести кратчайший путь из 3-й вершины во 2-ю. Элемент $M[3,2]$ равен 1, поэтому смотрим на элемент $M[3,1]$. Он равен четырем. Сравниваем $M[3,4]$ с 3-й. Есть совпадение, мы получили кратчайший путь: $3\to4\to1\to2$.

```
Procedure All_Way(i,j:Integer);{*Вывод пути между
        вершинами i и j.*}
```

```
Begin
 If M[i,j]=i Then
   If i=j Then Write(i) Else Write(i,'-',j)
              Else
     Begin
       All_Way(i,M[i,j]); All_Way(M[i,j],j);
     End;
End;
```

4.7. Независимые и доминирующие множества

Задача поиска подмножеств множества вершин V графа G, удовлетворяющих определенным условиям, свойствам, возникает достаточно часто.

4.7.1. Независимые множества

Дан неориентированный граф $G=(V,E)$. Независимое множество вершин есть множество вершин графа G, такое, что любые две вершины в нем не смежны, т. е. никакая пара вершин не соединена ребром. Следовательно, любое подмножество S, содержащееся в V, и такое, что пересечение S с множеством вершин смежных с S пусто, является независимым множеством вершин.

Пример.

Множества вершин (1, 2), (3, 4, 5), (4, 7), (5, 6) независимые (см. рисунок 4.30).

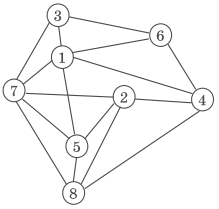


Рис. 4.30

Независимое множество называется максимальным, когда нет другого независимого множества, в которое оно бы входило. Если Q является семейством всех независимых множеств графа G, то число

$$a[G]=\max_{S\in Q}|S|$$

называется числом независимости графа G, а множество S^*, на котором этот максимум достигается, называется наибольшим независимым множеством. Для примера на рисунке 4.30 $a[G]=3$, а S^* есть (3, 4, 5).

Понятие, противоположное максимальному независимому множеству, есть максимальный полный подграф (клика). В максимальном независимом множестве нет смежных вершин, в клике все вершины попарно смежны. Максимальное независи-

мое множество графа G соответствует клике графа G', где G' — дополнение графа G.

Для примера на рисунке 4.30 дополнение G' приведено на рисунке 4.31(*а*), клика графа G'(рисунок 4.31(*б*)) соответствует максимальному независимому множеству графа G. Число независимости графа G' равно 4, максимальное независимое множество (2, 5, 7, 8), ему соответствует клика графа G (рисунок 4.31(*в*)).

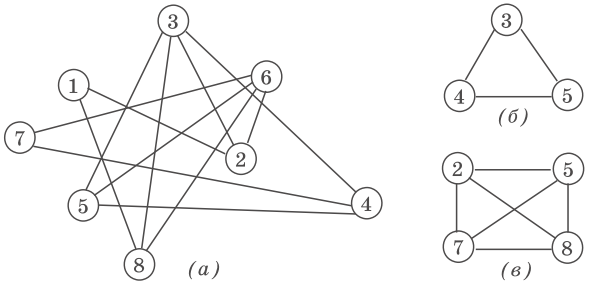


Рис. 4.31

4.7.2. Метод генерации всех максимальных независимых множеств графа

Задача решается перебором вариантов. «Изюминкой» алгоритма является отсутствие необходимости запоминать генерируемые множества с целью проверки их на максимальность путем сравнения с ранее сформированными множествами. Идея заключается в последовательном расширении текущего независимого множества (k — номер шага или номер итерации в процессе построения). Очевидно, что если мы не можем расширить текущее решение, то найдено максимальное независимое множество. Выведем его и продолжим процесс поиска.

Будем хранить текущее решение в массиве *Ss* (*Array*[1..*N*] *Of Integer*), его первые k элементов определяют текущее решение.

Алгоритм вывода решения очевиден (процедура *Print*).

В процессе решения нам придется многократно рассматривать вершины графа, смежные с данной. При описании графа матрицей смежности — это просмотр соответствующей строки, при описании списками связи — просмотр списка. Упростим нахождение смежных вершин за счет использования нового способа описания графа. Используем множественный тип данных. Введем тип данных:

```
Type Sset=Set Of 1..N
```

и переменную

```
Var A:Array[1..N] Of Sset.
```

Итак, чтобы определить вершины графа, смежные с вершиной i, необходимо просто вызвать соответствующий элемент массива A.

Пример. Основная сложность алгоритма в выборе очередной вершины графа. Введем переменную *Gg* для хранения номеров вершин — кандидатов на расширение текущего решения. Значение переменной формируется на каждом шаге k (см. рисунок 4.32). Что является исходной информацией для формирования *Gg*? Очевидно, что некоторое множество вершин, свое для каждого шага (итерации) алгоритма. Логически правомерно разбить это множество вершин на не использованные ранее (*Qp*) и использованные ранее (*Qm*). Изменение значений *Qp* и *Qm* происходит при возврате на выбор k-го элемента максимального независимого множества. Мы выбирали на k шаге, например, вершину с номером i, и естественно исключение ее из *Qp* и *Qm* при поиске следующего максимального независимого множества. Кроме того, при переходе к шагу с номером (k+1) из текущих множеств *Qp* и *Qm* для следующего шага необходимо исключить вершины, смежные с вершиной i, выбранной на данном шаге (из определения независимого множества), и, разумеется, саму вершину i.

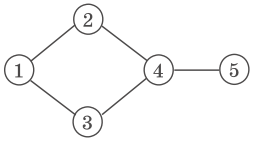


Рис. 4.32

Итак, общий алгоритм.

```
Procedure Find(k:Integer; Qp,Qm:Sset);
 Var Gg:Sset;
 i:Byte;
 Begin
  If (Qp=[ ]) And (Qm=[ ]) Then
    Begin Print(k); Exit; End;
      <формирование множества кандидатов Gg для
      расширения текущего решения (k элементов
      в массиве Ss) по значениям Qp и Qm - "черный
      ящик A".>;
```

```
i:=1;
While i<=N Do
  Begin
     If i In Gg Then
      Begin
        Ss[k]:=i;
        Find(k+1,Qp-A[i]-[i],Qm-A[i]-[i]);
       <изменение Qp, Qm для этого уровня
       (значения k)и, соответственно, изменение
       множества кандидатов Gg - "черный ящик Б".>;
      End;
     Inc(i);
  End;
End;
```

Продолжим уточнение алгоритма. Нам потребуется простая функция подсчета количества элементов в переменных типа *Sset*.

```
Function Number(A:Sset):Byte;
 Var i, cnt:Byte;
 Begin
  cnt:=0;
  For i:=1 To N Do
   If i In A Then Inc(cnt);
  Number:=cnt;
 End;
```

Используем метод трассировки для того, чтобы найти дальнейший алгоритм уточнения решения. Будем делать разрывы таблицы для внесения пояснений в работу черных ящиков (см. таблицу 4.4).

Таблица 4.4

k	*Qp*	*Qm*	*Gg*	*Ss*	Примечания
1	[1..5]	[ ]	[1..5]	(1)	Выбираем первую вершину
2	[4,5]	[ ]	[4,5]	(1,4)	Вершины 2, 3 исключаются из списка кандидатов, остаются вершины с номерами 4,5

Первое уточнение «черного ящика А».

```
If Qm<>[] Then <корректировка Gg с учетом ранее
                использованных вершин>
          Else Gg:=Qp;
```

Его суть в том, что если выбирать из ранее использованных вершин нечего, то множество кандидатов совпадает со значением *Qp*, и далее по алгоритму мы «тупо» выбираем первую вершину из *Qp*. Переходим к следующему вызову процедуры *Find* (см. таблицу 4.5).

Таблица 4.5

3	[ ]	[ ]	[ ]	(1,4)	Вывод первого максимального независимого множества и выход в предыдущую копию *Find*
2	[5]	[4]	[5]	(1,5)	Исключаем вершину 4 из *Qp* и включаем ее в *Qm*

Продолжает работу цикл *While* процедуры *Find* с параметром *k*, равным 2. Выбираем следующую вершину — это вершина 5. И вызываем процедуры *Find* с другими значениями параметров (см. таблицу 4.6).

Таблица 4.6

3	[ ]	[ ]	[ ]	(1,5)	Вывод второго максимального независимого множества
2	[ ]	[4,5]	[ ]		Цикл *While* закончен, выход в предыдущую копию процедуры *Find* (*k*=1)

Уточнение «черного ящика Б».

Первое: необходимо исключить вершину *i* из *Qp* и включить ее в *Qm*.

Второе: следует откорректировать множество *Gg*. Выбор на этом шаге вершин, не смежных с *i*, приведет к генерации повторяющихся максимальных независимых множеств, поэтому следует выбирать вершины из пересечения множеств *Qp* и *A*[*i*]. Итак, «черный ящик Б».

```
Qp:=Qp-[i];Qm:=Qm+[i];
Gg:=Qp*A[i];
```

Следующий шаг — выход в предыдущую версию *Find*, при этом значение *k*=1 (см. таблицу 4.7).

Таблица 4.7

1	[2..5]	[1]	[1..5]		Однако после работы «черного ящика Б» имеем следующие значения переменных
1	[2..5]	[1]	[2..3]	(2)	
2	[3,5]	[ ]	[3,5]	(2,3)	
3	[5]	[ ]	[5]	(2,3,5)	
4	[ ]	[ ]	[ ]		Вывод третьего максимального независимого множества
3	[ ]	[5]	[ ]		
2	[5]	[3]	[ ]		Согласно логике «черного ящика Б» множество кандидатов *Gg* становится пустым
1	[3..5]	[1,2]	[2,3]	(3)	
2	[5]	[2]			Итак, мы первый раз попадаем в процедуру *Find*, и множество *Gm* при этом не пусто

Должен работать алгоритм «черного ящика А» при непустом значении Gm.

Замечание 1. Если существует вершина j, принадлежащая Qm, такая, что пересечение $A[j]$ и Qp пусто, то дальнейшее построение максимального независимого множества бессмысленно — вершины из $A[j]$ не попадут в него (в нашем примере как раз этот случай).

Замечание 2. Если нет вершин из Qm, удовлетворяющих первому замечанию, то какую вершину из Qp следует выбирать? Ответ: вершину $i \in (Qp \cap A[j])$ для некоторого значения $j \in Qm$, причем мощность пересечения множеств $A[j]$ и Qp минимальна. Данный выбор позволяет сократить перебор.

Итак, окончательный алгоритм «черного ящика А».

```
Begin
 delt:=N+1;
 For j:=1 To N Do
  If j In Qm Then
   If Number(A[j]*Qp)<delt Then
    Begin
       i:=j; delt:=Number(A[j]*Qp);
    End;
  Gg:=Qp*A[i];
End
```

Закончим трассировку примера.

Таблица 4.8

2	[5]	[2]	[ ]		
1	[5]	[1,2,3]	[ ]		Выход в основную программу

Независимые максимальные множества графа найдены.

4.7.3. Доминирующие множества

Для графа $G=(V,E)$ доминирующее множество вершин есть множество вершин $S \subset V$, такое, что для каждой вершины j, не входящей в S, существует ребро, идущее из некоторой вершины множества S в вершину j. Доминирующее множество называется минимальным, если нет другого доминирующего множества, содержащегося в нем.

Пример.

Доминирующие множества (1, 2, 3), (4, 5, 6, 7, 8, 9), (1, 2, 3, 8, 9), (1, 2, 3, 7) и т. д. Множества (1, 2, 3), (4, 5, 6, 7, 8, 9) являются минимальными. Если Q — семейство всех минимальных доминирующих множеств графа, то число $\beta[G]=\min_{S \in Q} |S|$

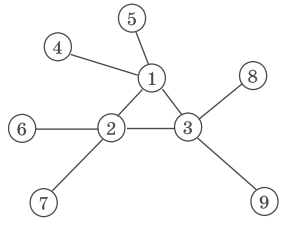
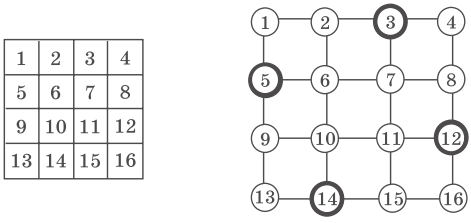


Рис. 4.33

называется числом доминирования графа, а множество S^*, на котором этот минимум достигается, называется наименьшим доминирующим множеством. Для примера на рисунке 4.33 $\beta[G]=3$.

Задача. Пусть город можно изобразить как квадрат, разделенный на 16 районов (см. рисунок 4.34). Считается, что опорный пункт милиции, расположенный в каком-либо районе, может контролировать не только этот район, но и граничащие с ним районы. Требуется найти наименьшее количество пунктов милиции и места их размещения, такие, чтобы контролировался весь город.

Представим каждый район вершиной графа и ребрами соединим только те вершины, которые соответствуют соседним районам. Нам необходимо найти число доминирования графа и хотя бы одно наименьшее доминирующее множество. Для данной задачи $\beta[G]=4$, и одно из наименьших множеств есть (3, 5, 12, 14). Эти вершины выделены на рисунке 4.34 (б).



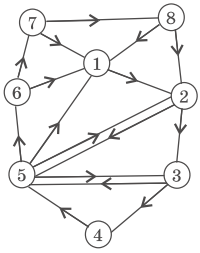
(а) *(б)*

Рис. 4.34

4.7.4. Задача о наименьшем покрытии

Рассмотрим граф. На рисунке 4.35 показана его матрица смежности A и транспонированная матрица смежности с единичными диагональными элементами A^*. Задача определения доминирующего множества графа G эквивалентна задаче нахождения такого наименьшего множества столбцов в матрице A^*, что каждая строка матрицы содержит единицу хотя бы в одном из выбранных столбцов. Задачу о поиске наименьшего множества столбцов, «покрывающего» все строки матрицы, называют задачей о *наименьшем покрытии*.

В общем случае матрица не обязательно является квадратной, кроме того, вершинам графа (столбцам) может быть приписан вес, в этом случае необходимо найти покрытие с наименьшей общей стоимостью. Если введено дополнительное ограничение, суть которого в том, чтобы любая пара столбцов не имела общих единиц в одних и тех же строках, то задачу называют задачей *о наименьшем разбиении*.



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad A^* = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Рис. 4.35

Предварительные замечания.

1. Если некоторая строка матрицы A^* имеет единицу в единственном столбце, т. е. больше нет столбцов, содержащих еди-

ницу в этой строке, то данный столбец следует включать в любое решение.

2. Рассмотрим множество столбцов матрицы A^*, имеющих единицы в конкретной строке.

Для нашего примера:

U^1=(1, 6, 7, 8),
U^2=(1, 2, 5, 8),
U^3=(2, 3, 5),
U^4=(3, 4),
U^5=(2, 3, 4, 5),
U^6=(5, 6),
U^7=(6, 7),
U^8=(7,8).

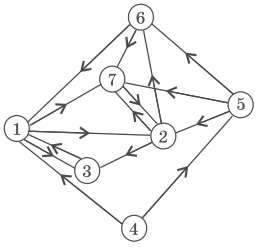
Видим, что $U^4 \subset U^5$. Из этого следует, что 5-ю строку можно не рассматривать, поскольку любое множество столбцов, покрывающее 4-ю строку, должно покрывать и 5-ю. Четвертая строка доминирует над пятой.

4.7.5. Метод решения задачи о наименьшем разбиении

Попытаемся осознать метод решения задачи, рассматривая, как обычно, пример.

У нас есть ориентированный граф, его матрица смежности и транспонированная матрица смежности с единичными диагональными элементами (см. рисунок 4.36). Исследуем структуру матрицы A^*. Нас интересует, какие столбцы содержат единицу в первой строке, какие столбцы содержат единицу во второй строке и не содержат в первой и так далее. С этой целью можно было бы переставлять столбцы в матрице A^*, но оставим ее «в покое». Будем использовать дополнительную матрицу *Bl*, ее тип:

```
Type Pr = Array [1..MaxN, 1..MaxN+1] Of Integer;
                Var Bl:Pr;
```



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad A^* = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Рис. 4.36

где *MaxN* — максимальная размерность задачи. Почему плюс единица (технический прием — «барьер»), будет ясно из последующего изложения (процедура *Press*).

При инициализации матрица *Bl* должна иметь вид: в первой строке — [1 2 3 .. *N* 0], а все остальные элементы равны нулю.

Наше исходное предположение заключается в том, что все столбцы матрицы *A** имеют единицы в первой строке. Проверим его. Будем просматривать элементы очередной строки (*i*) матрицы *Bl*. Если *Bl*[*i,j*]<>0, то со значением *Bl*[*i,j*], как номером столбца матрицы *A**, проверим соответствующий элемент *A**. При его неравенстве нулю элемент *Bl* остается на своем месте, иначе он переписывается в следующую строку матрицы *Bl*, а элементы текущей строки *Bl* сдвигаются вправо, сжимаются (*Press*). Итак, для *N*−1 строки матрицы *Bl*.

Для нашего примера матрица *Bl* после этого преобразования будет иметь вид:

$$Bl = \begin{pmatrix} 1 & 3 & 4 & 6 & 0 & \dots & 0 \\ 2 & 5 & 7 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

В задаче заданы стоимости вершин графа или стоимости столбцов матрицы *A**, и необходимо найти разбиение наименьшей стоимости. Пусть стоимости описываются в массиве *Price (Array[1..MaxN] Of Integer)* и для примера на рисунке 4.36 имеют значения [15 13 4 3 8 9 10]. Осталась чисто техническая деталь — отсортировать элементы каждой строки матрицы *Bl* по возрастанию стоимости соответствующих столбцов матрицы *A**. Результирующая матрица *Bl* имеет вид:

$$Bl = \begin{pmatrix} 4 & 3 & 6 & 1 & 0 & \dots & 0 \\ 5 & 7 & 2 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

а алгоритм формирования описан в процедуре *Blocs*.

```
   Procedure Blocs;{*Выделение блоков, Bl -
глобальная переменная.*}
    Procedure Sort;
     Begin
```

```
...
  End;
 Procedure Press(i,j:Integer);{*Сдвигаем элементы
          строки с номером i, начиная с позиции
          (столбца) j, на одну позицию вправо.*}
  Var k:Integer;
  Begin
   k:=j;
   While Bl[i,k]<>0 Do
    Begin{*Поэтому размерность
          матрицы с плюс единицей. В последнем
          столбце строки всегда записан 0.*}
       Bl[i,k]:=Bl[i,k+1];
       Inc(k);
    End;
  End;

Var i,j,cnt:Integer;
Begin
 FillChar(Bl,SizeOf(Bl),0);
 For i:=1 To N Do Bl[1,i]:=i;{*Предполагается, что
      в первом блоке все столбцы.*}
 For i:=1 To N-1 Do
  Begin
   j:=1;cnt:=0;
   While Bl[i,j]<>0 Do
    Begin
     If A*[i,Bl[i,j]]=0 Then
      Begin{*Столбец не в этом блоке.*}
       Inc(cnt);
       Bl[i+1,cnt]:=Bl[i,j];{*Переписать в следующую
                             строку.*}
       Press(i,j);
       Dec(j);
      End;
     Inc(j);
    End;
  End;
 Sort;
End;
```

Изменим чуть-чуть A^* ($A^*[2,7]=0$, $A^*[4,7]=1$). Какой вид будет иметь матрица Bl?

После этой предварительной работы мы имеем вполне «приличную» организацию данных для решения задачи путем перебора вариантов. Матрица *Bl* разбита на блоки, и необходимо выбрать по одному элементу (если соответствующие строки еще не покрыты) из каждого блока. Процесс выбора следует продолжать до тех пор, пока не будут включены в «покрытие» все строки или окажется, что некоторую строку нельзя включить.

Продолжим рассмотрение метода. Если при поиске независимых множеств мы шли «сверху вниз», последовательно уточняя логику, то сейчас попробуем идти «снизу вверх», складывая окончательное решение из сделанных «кирпичиков».

Как обычно, следует начать со структур данных. Во-первых, мы ищем лучшее решение, т. е. то множество столбцов, которое удовлетворяет условиям задачи (не пересечение и «покрытие» всего множества строк), и суммарная стоимость этого множества минимальна. Значит, необходима структура данных для хранения этого множества и значения наилучшей стоимости и, соответственно, структуры данных для хранения текущего (очередного) решения и его стоимости. Во-вторых, в решении строка может быть или не быть. Следовательно, нам требуется как-то фиксировать эту информацию. Итак, данные.

```
Type Model=Array[1..MaxN] Of Boolean;
Var Sbetter:Model;Pbetter:Integer;{*Лучшее
      решение.*}
    S:Model;P:Integer;{*Текущее решение.*}
    R:Model;{*R[i]=True - признак того, что
         строка i "покрыта" текущим решением.*}
```

Алгоритм включения (исключения) столбца с номером k в решение (из решения) имеет вид:

```
Procedure Include(k:Integer); {*Включить столбец
        в решение.*}
 Var j:Integer;
 Begin
  P:=P+Price[k];{*Текущая цена решения.*}
  S[k]:=True;{*Столбец с номером k в решение.*}
  For j:=1 To N Do
   If A*[j,k]=1 Then R[j]:=True; {*Строки,
          "покрытые" столбцом k.*}
 End;
Procedure Exclude(k:Integer);{*Исключить столбец
          из решения.*}
 Var j:Integer;
```

```
Begin
 p:=p-Price[k];
 S[k]:=False;
 For j:=1 To N Do
  If (A*[j,k]=1) And R[j] Then R[j]:=False;
End;
```

Проверка, сформировано ли решение, заключается в том, чтобы просмотреть массив R и определить, все ли его элементы равны истине.

```
Function Result:Boolean;
 Var j:Integer;
 Begin
  j:=1;
  While (j<=N) And R[j] Do Inc(j);
  Result:=j=N+1;
 End;
```

Кроме перечисленных действий, нам необходимо уметь определять — «можно ли столбец с номером k включать в решение». Для этого следует просмотреть столбец с номером k матрицы A^* и проверить, нет ли совпадений единичных элементов со значением *True* соответствующих элементов массива R.

```
Function Cross(k:Integer):Boolean; {*Пересечение
      столбца с частичным решением, сформированным
      ранее.*}
 Var j:Integer;
 Begin
  j:=1;
  While (j<=N) And Not(R[j] And (A*[j,k]=1)) Do Inc(j);
  Result:=j=N+1;
 End;
```

Заключительная процедура поиска *Find* имеет в качестве параметров номер блока (строки матрицы *Bl*) — переменная *bloc* и номер позиции в строке. Первый вызов — *Find(1,1)*.

```
Procedure Find(bloc,jnd:Integer);
 Begin
  If Result Then
    Begin
     If P<Pbetter Then
      Begin
        Pbetter:=P;
```

```
      Sbetter:=S;
    End;
   End
   Else
If Bl[bloc,jnd]=0
     Then Exit
     Else
      If Cross(Bl[bloc,jnd])
       Then
        Begin
         Include(Bl[bloc,jnd]);
         Find(bloc+1,1);
         Exclude(Bl[bloc,jnd]);
        End
       Else
           If R[bloc] Then Find(bloc+1,1)
                      Else Find(bloc,jnd+1);
End;
```

Будет ли правильно работать процедура *Find* при измененной матрице *A** (изменение приведено выше по тексту)? Исправьте неточность.

Нам осталось привести основную программу, но после выполненной работы она не вызывает затруднений.

```
Program R_min;
 Const MaxN=...;
 Type ...
 Var ...
 Procedure Init; {*Ввод и инициализация данных.*}
  Begin
  ...
  End;
Procedure Print;{*Вывод результата.*}
 Begin
 ...
 End;
{*Процедуры и функции, рассмотренные ранее по тексту.*}
 Begin {*Основная логика.*}
  Init;
  Blocs;
  Find(1,1);
  Print;
 End.
```

4.8. Раскраски

4.8.1. Правильные раскраски

Пусть $G=(V,E)$ — неориентированный граф.

Произвольная функция $f:V\to\{1,2,\ldots,k\}$, где k принадлежит множеству натуральных чисел, называется вершинной k-раскраской графа G.

Раскраска называется правильной, если $f(u)\neq f(v)$, для любых смежных вершин u и v.

Граф, для которого существует правильная k-раскраска, называется k-раскрашиваемым. Минимальное число k, при котором граф G является k-раскрашиваемым, называется хроматическим числом графа и обозначается $\chi(G)$.

Пример. $\chi(G)=3$. Меньшим количеством цветов граф правильно раскрасить нельзя из-за наличия треугольников. Рядом с «кружками» — вершинами графа — указаны номера цветов (см. рисунок 4.37).

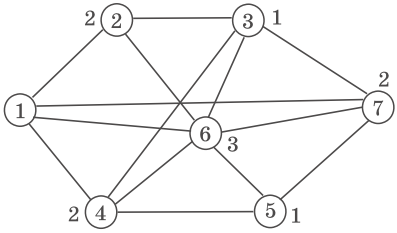


Рис. 4.37

Метод получения правильной раскраски достаточно прост — закрашиваем очередную вершину в минимально возможный цвет.

Введем следующие структуры данных.

```
Const Nmax=100; {*Максимальное количество вершин
                  графа.*}
Type V=0..Nmax;
       Ss=Set Of V;
       MyArray=Array[1..Nmax] Of V;
       Var Gr:MyArray;{*Gr - каждой вершине графа
             определяется номер цвета.*}
```

Для примера, приведенного на рисунке 4.37, массив Gr имеет вид: $Gr[1]=Gr[3]=Gr[5]=1$; $Gr[2]=Gr[4]=Gr[7]=2$; $Gr[6]=3$.

Фрагмент основного алгоритма.

```
....
<формирование описания графа>;
For i:=1 To N Do Gr[i]:=[Color(i,0)];
<вывод решения>;
```

Поиск цвета раскраски для одной вершины можно реализовать с помощью следующей функции:

```
Function Color(i,t:V):Integer;{*i - номер
     окрашиваемой вершины, t - номер цвета,
     с которого следует искать раскраску данной
     вершины, A - матрица смежности, Gr -
     результирующий массив.*}
 Var Ws:Ss;
     j:Byte;
 Begin
  Ws:=[ ];
  For j:=1 To i-1 Do
   If A[j,i]=1 Then Ws:=Ws+[Gr[j]];
         {*Формируем множество цветов, в которые
         окрашены смежные вершины с меньшими
         номерами.*}
  j:=t;
  Repeat {*Поиск минимального номера цвета,
       в который можно окрасить данную вершину.*}
    Inc(j);
  Until Not(j In Ws);
  Color:=j;
 End;
```

Пример. Для графа на рисунке 4.38 получаем правильную раскраску: *Gr*[1]=1, *Gr*[2]=*Gr*[4]=2, *Gr*[3]=3, *Gr*[5]=4. Однако минимальной раскраской является: *Gr*[1]=1, *Gr*[2]=*Gr*[5]=2 и *Gr*[3]=*Gr*[4]=3, и хроматическое число графа равно трем.

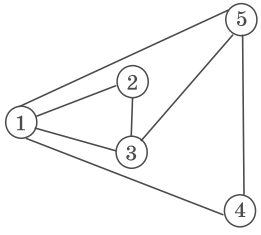


Рис. 4.38

4.8.2. Поиск минимальной раскраски вершин графа

Метод основан на простой идее [16], и в некоторых случаях он дает точный результат.

Пусть получена правильная раскраска графа, q — количество цветов в этой раскраске. Если существует раскраска, использующая только $q-1$ цветов, то все вершины, окрашенные в цвет q, должны быть окрашены в цвет g, меньший q.

Согласно алгоритму формирования правильной раскраски вершина была окрашена в цвет q, потому что не могла быть окрашена в цвет с меньшим номером. Следовательно, необходимо попробовать изменить цвет у вершин, смежных с рассматриваемой. Но как это сделать?

Найдем вершину с минимальным номером, окрашенную в цвет q, и просмотрим вершины, смежные с найденной. Попытаемся окрашивать смежные вершины не в минимально возможный цвет. Для этого находим очередную смежную вершину и стараемся окрасить ее в другой цвет. Если это получается, то перекрашиваем вершины с бóльшими номерами по методу правильной раскраски. При неудаче, когда изменить цвет не удалось или правильная раскраска не уменьшила количества цветов, переходим к следующей вершине. И так, пока не будут просмотрены все смежные вершины.

Опишем алгоритм, используя функцию *Color* предыдущего параграфа.

```
Var MaxC,MinNum,i:V;
Procedure MaxMin(Var c,t:V);
 Begin
...
 End;
Procedure Change(t:V);
 Begin
...
 End;
Begin
 <ввод данных, инициализация переменных>;
 For i:=1 To N Do Gr[i]:=Color(i,0);{*Первая
     правильная раскраска.*}
 MaxC:=0;MinNum:=0;
 Repeat
  <найти вершину с минимальным номером(MinNum),
       имеющую максимальный цвет раскраски(MaxC) -
       процедура MaxMin(MaxC,MinNum)>;
  <изменить цвет раскраски в соответствии
       с описанной идеей - процедура
      Change(MinNum)>;
```

```
 Until MaxC=Gr[MinNum];{*До тех пор, пока метод
       улучшает раскраску.*}
 <вывод минимальной (или почти минимальной!)
       раскраски>;
End.
```

Если работа процедуры *MaxMin* достаточно очевидна, то *Change* требует дальнейшего уточнения.

```
Procedure Change(t:V);
 Var r,q:V;
     Ws:Ss;
 Function MaxCon(l:V;Rs:Ss):V;{*Находим смежную
 с l вершину с наибольшим номером и
 не принадлежащую множеству Rs.*}
  Var i,w:V;
  Begin
   i:=l-1; w:=0;
   While (i>=1) And (i In Rs) And (w=0) Do
    Begin
       If A[l,i]=1 Then w:=i;
       Dec(i);
    End
   MaxCon:=w;
  End;
 Begin
  Ws:=[];
  q:=MaxCon(t,Ws);
  While q<>0 Do
   Begin
       r:=Color(q,Gr[q]);
       If r<MaxC Then <изменить цвет у вершин
          с номерами, большими t, по методу
          правильной раскраски>
                 Else Ws:=Ws+[q];
       q:=MaxCon(t,Ws);
   End;
 End;
```

Осталось заметить, что при изменении цвета у вершин с номерами, большими значения t, по методу правильной раскраски следует запоминать в рабочих переменных значения *Gr* и *MaxC*, так как при неудаче (раскраску не улучшим) их необходимо восстанавливать, и на этом закончить уточнение логики.

При любом упорядочении вершин допустимые цвета j для вершины с номером i удовлетворяют условию $j \le i$. Это очевидно, так как вершине i предшествует $i-1$ вершина, и, следовательно, никакой цвет $j > i$ не использовался.

Итак, для вершины 1 допустимый цвет 1, для 2 — цвет 1 и 2 и так далее.

С точки зрения скорости вычислений вершины следует помечать (присваивать номера) так, чтобы первые q вершин образовывали наибольшую клику графа G. Это приведет к тому, что каждая из этих вершин имеет один допустимый цвет и процесс возврата в алгоритме можно будет заканчивать при достижении вершины из этого множества.

В алгоритме рассматриваются только вершины, смежные с вершиной, окрашенной в максимальный цвет. Однако следует работать и с вершинами, являющимися смежными для смежных.

На рисунке 4.39 приведены примеры графов и их раскраски.

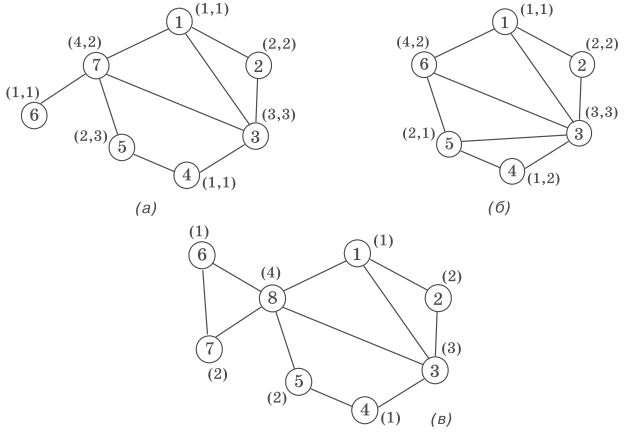


Рис. 4.39

Первая цифра в круглых скобках обозначает значение цвета при правильной раскраске, вторая — при минимальной.

ля графов на рисунках 4.39(а) и 4.39(в) алгоритм дает пра - вильный результат. Для графа на рисунке 4.39(б) при изменении цвета вершины с номером 6 необходимо просматривать вершины, смежные к смежным с шестой вершиной.

Рассмотренный алгоритм требует очередной модификации.

Самый интересный случай на рисунке 4.40. Мы не можем получить минимальную раскраску с помощью рассмотренного

алгоритма (даже если будем рассматривать все вершины, смежные к смежным), хотя она, конечно, существует и совпадает с предыдущей. Итак, результат работы алгоритма зависит от нумерации вершин графа, а таких способов — способов нумерации $N!$.

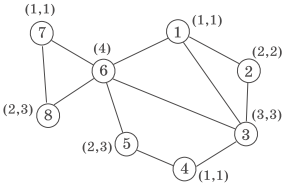


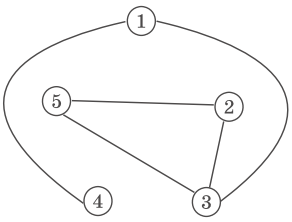
Рис. 4.40

4.8.3. Использование задачи о наименьшем покрытии при раскраске вершин графа

При любой правильной раскраске графа G множество вершин, окрашиваемых в один и тот же цвет, является независимым множеством. Поэтому раскраску можно понимать как разбиение вершин графа на независимые множества. Если рассматривать только максимальные независимые множества, то раскраска — это не что иное, как покрытие вершин графа G множествами этого типа. В том случае, когда вершина принадлежит не одному максимальному независимому множеству, допустимы различные раскраски с одним и тем же количеством цветов. Эту вершину можно окрашивать в цвета тех множеств, которым она принадлежит.

Исходным положением метода является получение всех максимальных независимых множеств и хранение их в некоторой матрице $M(N\times W)$, где W — количество максимальных независимых множеств. Элемент матрицы $M[i,j]$ равен единице, если вершина с номером i принадлежит множеству с номером j, и нулю в противном случае. Если затем каждому столбцу поставить в соответствие единичную стоимость, то задача раскраски не что иное, как поиск минимального количества столбцов в матрице M, покрывающих все ее строки. Каждый столбец соответствует определенному цвету.

Пример. На рисунке 4.41 представлено пять максимальных независимых множеств и два варианта решения задачи о покрытии (строки A и B). В обоих случаях вершина 4 может быть окрашена как во второй, так и в третий цвета.



	1	2	3	4	5
1	**1**	**1**	**0**	**0**	**0**
2	**1**	**0**	**0**	**1**	**0**
3	**0**	**0**	**1**	**0**	**0**
4	**0**	**0**	**1**	**1**	**1**
5	**0**	**1**	**0**	**0**	**1**
A	*		*		*
B		*	*	*	

Рис. 4.41

4.9. Потоки в сетях, паросочетания

4.9.1. Постановка задачи

Одной из задач теории графов является задача определения максимального потока, протекающего от некоторой вершины s графа (источника) к некоторой вершине t (стоку). При этом каждой дуге (граф ориентированный) (i,j) приписана некоторая пропускная способность $C(i,j)$, определяющая максимальное значение потока, который может протекать по данной дуге.

Содержательных интерпретаций задачи достаточно много, и, безусловно, они усилят и сделают более понятными сложные занятия по этой проблематике.

Метод решения задачи о максимальном потоке от s к t был предложен Фордом и Фалкерсоном, и их *«техника меток»* составляет основу других алгоритмов решения многочисленных задач, являющихся обобщениями или расширениями указанной задачи.

Одним из фундаментальных фактов теории потоков в сетях является классическая теорема о максимальном потоке и минимальном разрезе. Разрезом называют множество дуг, удаление которых из сети приводит к «разрыву» всех путей, ведущих из s в t. Пропускная способность разреза — это суммарная пропускная способность дуг, его составляющих. Разрез с минимальной пропускной способностью называют минимальным разрезом.

Теорема (Форд и Фалкерсон). Величина каждого потока из s в t не превосходит пропускной способности минимального разреза, разделяющего s и t, причем существует поток, достигающий этого значения.

Теорема устанавливает эквивалентность задач нахождения максимального потока и минимального разреза, однако не определяет метода их поиска.

Пример. На рисунке 4.42 показана сеть, источник — вершина 1, сток — вершина 6, в скобках у дуг указаны их пропускные способности. Минимальный разрез — дуги (1, 2) и (3, 4), следовательно, согласно теореме максимальный поток равен 4. Разрез определен путем простого перебора. Алгоритм его «лобового» поиска очевиден. Осуществляем перебор по дугам путем генерации

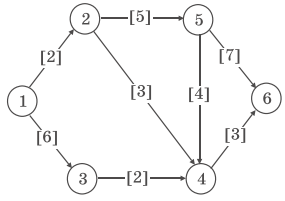


Рис. 4.42

всех возможных подмножеств дуг. Для каждого подмножества дуг проверяем, является ли оно разрезом. Если является, то вычисляем его пропускную способность и сравниваем ее с минимальным значением. При положительном результате сравнения запоминаем разрез и изменяем значение минимума.

Удачный выбор данных позволяет сделать программный код компактным, но очевидно, что даже при наличии различных отсечений в переборе метод применим только для небольших сетей. Однако, как найти максимальный поток, т. е. его распределение по дугам, по-прежнему открытый вопрос.

«Техника меток» Форда и Фалкерсона заключается в последовательном (итерационном) построении максимального потока путем поиска на каждом шаге увеличивающейся цепи, то есть пути (последовательности дуг), поток по которой можно увеличить. При этом узлы (вершины графа) специальным образом помечаются. Отсюда и возник термин «метка».

Пример из работы [9]. На рисунке 4.43 рядом с пропускными способностями дуг указаны потоки, построенные на этих дугах. На рисунке поток через сеть равен 10 и найдена увеличивающаяся цепочка, выделенная двойными линиями. Обратите внимание на ориентацию дуг, входящих в цепочку. По данной цепочке можно пропустить поток, равный 1, пропускная способность дуги (5, 6). Изменяем суммарный поток, его значение становится равным 11. Поток увеличен, необходимо продолжить поиск увеличивающихся цепочек; если окажется, что построить их нельзя, то результирующий поток максимален. Заметим, что для данного примера это значение потока окончательное. Обратите внимание на то, как изменен поток на дугах сети в зависимости от их ориентации.

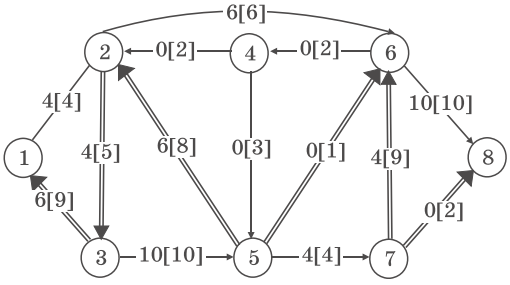


Рис. 4.43

4.9.2. Метод построения максимального потока в сети

Рассмотрим метод на примере рисунка 4.44.

Пусть дана сеть $G=(V,E)$, узлом-источником является вершина 1, узлом-стоком — вершина 6. Построим максимальный поток (F) между этими вершинами. Начальное значение F нулевое.

Очевидно, что структурой данных для описания F является матрица того же типа, что и матрица C, в которой определены пропускные способности дуг.

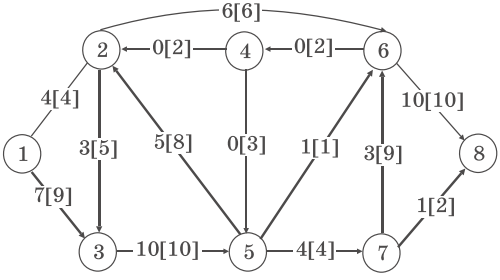


Рис. 4.44

Первая итерация (см. рисунок 4.45). Присвоим вершине 1 метку [1,@].

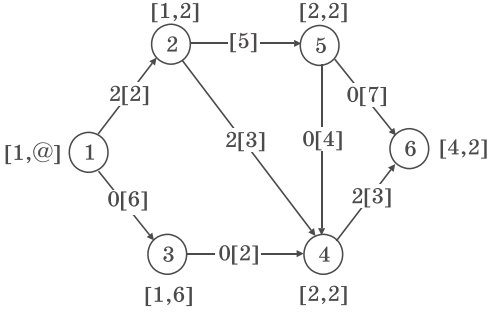


Рис. 4.45

Шаг 1. Рассмотрим дуги, началом которых является вершина 1 — дуги (1,2) и (1,3). Вершины 2 и 3 не помечены, поэтому присваиваем им метки, для 2-й — [1,2] и 3-й — [1,6]. Что представляют из себя метки? Первая цифра — номер вершины, из которой идет поток, вторая цифра — численное значение потока, который можно передать по этой дуге.

Шаг 2. Выберем помеченную, но непросмотренную вершину. Первой в соответствующей структуре данных записана вершина 2. Рассмотрим дуги, для которых она является началом — дуги (2,4) и (2,5). Вершины 4 и 5 не помечены. Присвоим им метки — [2,2] и [2,2]. Итак, на втором шаге вершина 2 просмотрена, вершины 3, 4, 5 помечены, но не просмотрены, остальные вершины не помечены.

Шаг 3. Выбираем вершину 3. Рассмотрим дугу (3,4). Вершина 4 помечена. Переходим к следующей вершине — четвертой, соответствующая дуга — (4,6). Вершина 6 не помечена. Присваиваем ей метку [4,2]. Мы достигли вершины-стока, тем самым найдя путь (последовательность дуг), поток по которым можно увеличить. Информация об этом пути содержит метки вершин.

В данном случае путь или увеличивающаяся цепочка 1→2→4→6. Максимально возможный поток, который можно передать по дугам этого пути, определяется второй цифрой метки вершины стока, то есть 2. Поток в сети стал равным 2.

Вторая итерация (см. рисунок 4.46).

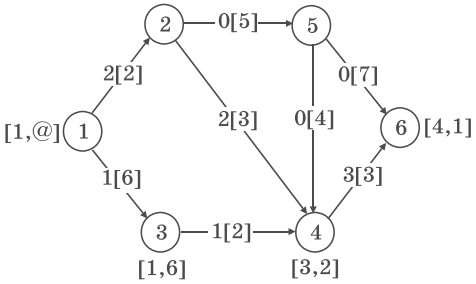


Рис. 4.46

Шаг 1. Присвоим вершине 1 метку [1,@]. Рассмотрим дуги, началом которых является помеченная вершина 1. Это дуги (1,2) и (1,3). Вершина 2 не может быть помечена, так как пропускная способность дуги (1,2) исчерпана. Вершине 3 присваиваем метку [1,6].

Шаг 2. Выберем помеченную, но не просмотренную вершину. Это вершина 3. Повторяем действия. В результате вершина 4 получает метку [3,2].

Шаг 3. Выбираем вершину 4, только она помечена и не просмотрена. Вершине 6 присваиваем метку [4,1].

Почему только одна единица потока? На предыдущей итерации израсходованы две единицы пропускной способности данной дуги, осталась только одна. Вершина-сток достигнута. Найдена увеличивающая поток цепочка, это 1→3→4→6, по которой можно «протащить» единичный поток. Результирующий поток в сети равен 3.

Третья итерация (см. рисунок 4.47). Вершине 1 присваиваем метку **[1,@]**.

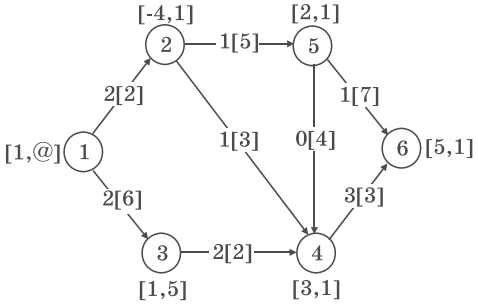


Рис. 4.47

Шаг 1. Результат — метка **[1,5]** у вершины 3.

Шаг 2. Метка **[3,1]** у вершины 4.

Шаг 3. Пропускная способность дуги (4,6) израсходована полностью.

Однако есть *обратная* дуга (2,4), по которой передается *поток, не равный нулю* (обратите внимание на текст, выделенный курсивом — «изюминка» метода). Попробуем перераспределить поток. Нам необходимо передать из вершины 4 поток, равный единице (зафиксирован в метке вершины). Задержим единицу потока в вершине 2, то есть вернем единицу потока из вершины 4 в вершину 2. Эту особенность зафиксируем в метке вершины 2 — [−4,1]. Тогда единицу потока из вершины 4 мы передадим по сети вместо той, которая задержана в вершине 2, а единицу потока из вершины 2 попытаемся «протолкнуть» по сети, используя другие дуги. Итак, вершина 4 просмотрена, вершина 2 помечена, вершины 5 и 6 не помечены.

Шаг 4 и **5** очевидны. Передаем единицу потока из вершины 2 в вершину 6 через вершину 5. Вершина-сток достигнута, найдена цепочка 1→3→4→2→5→6, по которой можно передать поток, равный единице. При этом по прямым дугам поток увеличивается на единицу, по обратным — уменьшается. Суммарный поток в сети — 4 единицы.

Четвертая итерация. Вершине 1 присваиваем метку [1,@].

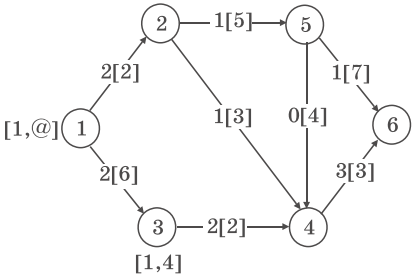


Рис. 4.48

Шаг 1. Помечаем вершину 3 — [1,4].

Шаг 2. Рассматриваем помеченную, но не просмотренную вершину 3. Одна дуга — (3,4). Вершину 4 пометить не можем — пропускная способность дуги исчерпана. Помеченных вершин больше нет, и вершина-сток не достигнута. Увеличивающую поток цепочку построить не можем. Найден максимальный поток в сети. Можно заканчивать работу.

Итак, в чем суть «техники меток» Форда и Фалкерсона?

1. На каждой итерации вершины сети могут находиться в одном из трех состояний: вершине присвоена метка, и она просмотрена; вершине присвоена метка, и она не просмотрена, то есть не все смежные с ней вершины обработаны; вершина не имеет метки.

2. На каждой итерации мы выбираем помеченную, но не просмотренную вершину v и пытаемся найти вершину u, смежную с v, которую можно пометить. Помеченные вершины, достижимые из вершины-источника, образуют множество вершин сети G. Если среди этих вершин окажется вершина-сток, то это означает успешный результат поиска цепочки, увеличивающей поток, при неизменности этого множества работа заканчивается — поток изменить нельзя.

Входные данные.

Сеть, описываемая $G=(V,E)$ матрицей пропускных способностей $C[1..N,1..N]$, где N — количество вершин.

Вершина-источник s и вершина-сток t.

Выходные данные.

Поток, описываемый матрицей $F[1..N,1..N]$.

Рабочие переменные.

Структура данных для хранения меток — $P[1..N,1..2]$.

Элемент *P*[*i*,1] — номер вершины, из которой можно передать поток, равный *P*[*i*,2], в вершину с номером *i*.

Логическая переменная *Lg*, значение *True* — есть цепочка, увеличивающая поток, *False* — нет.

Основной алгоритм.

```
Begin
 <ввод данных и инициализация переменных(Lg:=True)>;
 While Lg Do
Begin
   FillChar(P,SizeOf(P),0);
   <процедура расстановки меток(Mark), если
 вершину t не смогли  пометить, то Lg:=False;
 результат работы - значение P (метки вершин) >;
   If Lg Then <процедура Stream(t) - изменение
 потока по дугам найденной цепочки от
 вершины-стока t до вершины-источника s; входные
 данные - массив P, результат - измененный массив
 F>;
 End;
 <вывод потока F>;
End.{конец обработки}
```

Уточним алгоритм расстановки меток (не лучший вариант).

```
Procedure Mark;
 Var M:Set Of 1..N;
    i,l:integer;
 Begin
  M:=[1..N]; {*Непросмотренные вершины.*}
  P[s,1]:=s;P[s,2]:=maxint;{*Присвоим метку
        вершине-источнику.*}
  l:=s;
  While (P[t,1]=0) And Lg Do
    Begin
     For i:=1 To N Do {*Поиск непомеченной вершины.*}
      If (P[i,1]=0) And ((C[l,i]<>0) Or (C[i,l]<>0))
        Then
         If F[l,i]<C[l,i] Then
           Begin{*Дуга прямая?*}
            P[i,1]:=l;
            If P[l,2]<C[l,i]-F[l,i]
             Then P[i,2]:=P[l,2]
             Else P[i,2]:=C[l,i]-F[l,i];
           End
```

```
    Else
      If F[i,l]>0 Then
        Begin{*Дуга обратная?*}
          P[i,1]:=-l;
          If P[l,2]<F[i,l]
            Then P[i,2]:=P[l,2]
            Else P[i,2]:=F[i,l];
        End;
    M:=M-[l];{*Вершина с номером l просмотрена.*}
    l:=1;{*Находим помеченную и непросмотренную
          вершину.*}
    Repeat
     Inc(l)
    Until (l>N) Or ((P[l,1]<>0) And (l In M));
    If l>N Then Lg:=False;
  End;
End;
```

Алгоритм изменения потока F имеет вид:

```
Procedure Stream(q:Integer);
 Begin {*Определяем тип дуги - прямая или обратная,
       знак минус у номера вершины - признак обратной
       дуги.*}
  If P[q,1]>0
    Then F[P[q,1],q]:=F[P[q,1],q]+P[t,2]
    Else F[q,abs(P[q,1])]:=F[q,abs(P[q,1])]-P[t,2];
  If Abs(P[q,1])<>s Then
    Begin
      {*Если не вершина-источник, то переход к
      предыдущей вершине цепочки.*}
      q:=Abs(P[q,1]);
      Stream(q);
    End;
 End;
```

Итак, рассмотрение метода построения максимального потока в сети завершено.

4.9.3. Наибольшее паросочетание в двудольном графе

Паросочетанием в неориентированном графе $G=(V,E)$ называется произвольное множество ребер $M \subseteq E$, такое, что никакие два ребра из M не инцидентны одной вершине.

Вершины в G, не принадлежащие ни одному ребру паросочетания, называют свободными.

Граф G называют двудольным, если его множество вершин можно разбить на непересекающиеся множества — $V=X\cup Y$, $X\cap Y=\varnothing$, причем каждое ребро $e\in E$ имеет вид $e=(x,y)$, где $x\in X$, $y\in Y$. Двудольный граф будем обозначать $G=(X,Y,E)$.

Задача нахождения наибольшего паросочетания в двудольном графе сводится к построению максимального потока в некоторой сети. Рассмотрим схему сведения. На рисунке 4.49 показан исходный двудольный граф G.

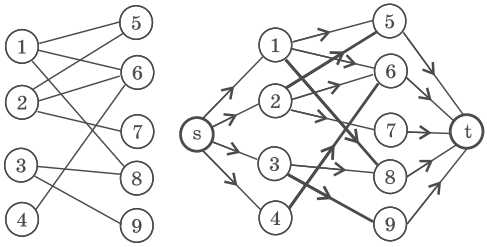


Рис. 4.49

Построим сеть $S(G)$ с источником s и стоком t следующим образом:

- источник s соединим дугами с вершинами из множества X;
- вершины из множества Y соединим дугами со стоком t;
- направление на ребрах исходного графа будет от вершин из X к вершинам из Y;
- пропускная способность всех дуг $C[i,j]=1$.

Найдем максимальный поток из s в t для построенной сети. Он существует [18]. Насыщенные дуги потока (они выделены «жирными» линиями на рисунке) соответствуют ребрам наибольшего паросочетания исходного графа. Найденное наибольшее паросочетание не единственное. Проверьте, это ли паросочетание получается при помощи алгоритма нахождения максимального потока. Найдите другие паросочетания.

Рассмотрим другой метод построения наибольшего паросочетания в двудольном графе. Введем понятие *чередующейся цепи* из X в Y относительно данного паросочетания M. Произвольное множество дуг $P\subseteq E$ вида:

$$P=\{(x_0,y_1),\ (y_1,x_1),\ (x_1,y_2),\ \ldots,\ (y_k,x_k),\ (x_k,y_{k+1})\},$$

где $k>0$,
все вершины различны,

x_0 — свободная вершина в X,
y_{k+1} — свободная вершина в Y,
каждая вторая дуга принадлежит M, то есть

$$P \cap M = \{(y_1,x_1),\ (y_2,x_2),\ ...,\ (y_k,x_k)\}.$$

Теорема [18]. Паросочетание M в двудольном графе G наибольшее тогда и только тогда, когда в G не существует чередующейся цепи относительно M.

Теорема подсказывает реальный путь нахождения наибольшего паросочетания. Пусть найдено некоторое паросочетание в графе G, на рисунке 4.50 оно выделено «жирными» линиями. Ищем чередующуюся цепь — ее дуги выделены линиями со стрелками. Она состоит из «тонких» дуг и «жирных», причем начинается и заканчивается в свободных вершинах. Длина цепи нечетна. Делаем «тонкие» дуги «жирными» и наоборот. Паросочетание увеличено на единицу.

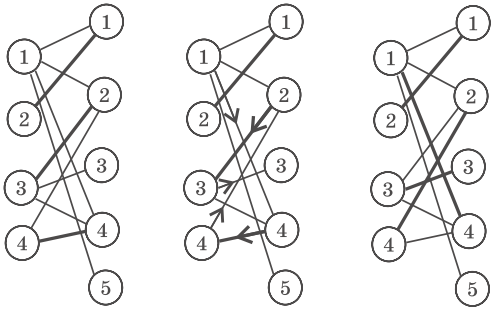


Рис. 4.50

Общий алгоритм.

```
Begin
 <ввод и инициализация данных>;
 <найти первое паросочетание>;
 While <есть чередующаяся цепочка> Do <изменить
      паросочетание>;
 <вывод результата>;
End.
```

Очередь за определением данных и последовательным уточнением алгоритма.

Граф описывается матрицей $A[N,M]$, где N — количество вершин в множестве X, а M — в Y.

Паросочетание будем хранить в двух одномерных массивах *XN* и *YM*. Для примера на рисунке 4.50 *XN*=[0,1,2,4] и *YM*=[2,3,0,4,0].

Уточнение фрагмента «найти первое паросочетание» не требует разъяснений. Можно начать и с одного ребра в первом паросочетании, но в этом случае количество итераций в основном алгоритме возрастает.

```
For i:=1 To N Do
Begin {*Назначение переменных i,
     j, pp очевидно.*}
  j:=1; pp:=True;
  While (j<=M) And pp Do
    Begin
      If (A[i,j]=1) And (YM[j]=0) Then
        Begin
          XN[i]:=j;YM[j]:=i;
          pp:=False;
        End;
      Inc(j);
    End;
End;
```

Как лучше хранить чередующуюся цепочку, учитывая требование изменения паросочетания и продумывая алгоритм ее построения? Естественно, массив, пусть это будет *Chain:*

Array[1..*NMmax*] *of NMmax*.. *NMmax*,

где *NMmax* — максимальное число вершин в графе, и его значение для рассмотренного примера равно [1, −4, 4, −2, 3, −3], т. е. вершины из множества *YM* хранятся со знаком минус (вспомните метод построения максимального потока).

Итак, поиск чередующейся цепочки.

```
Function FindChain:Boolean;
 Var p,yk,r:Word;
 Begin
  <инициализация данных, в частности yk:=1;>;
  <нахождение свободной вершины в XN>;
  If <вершина не найдена>
    Then FindChain:=False
    Else
      Begin
        p:=<номер первой свободной вершины>;
        Chain[yk]:=p;
```

```
    Repeat
     r:=<очередная вершина из Chain>;
     For <для всех вершин, связанных с r> Do
      If <вершина из XN>
       Then
          Begin
           If <ребро "тонкое"> Then <включить
                          ребро в цепочку>
          End
       Else
          Begin
            If <ребро "толстое"> Then
                      <включить ребро в цепочку>
          End;
    Until <просмотрены все вершины из Chain> or
              <текущая вершина принадлежит YM,
              и она свободна>;
    FindChar:=<текущая вершина принадлежит YM,
              и она свободна>;
  End;
 End;
```

Процесс дальнейшего уточнения алгоритма вынесем в самостоятельную часть работы.

4.10. Методы приближенного решения задачи коммивояжера

4.10.1. Метод локальной оптимизации

Попытаемся отказаться от перебора вариантов при решении задачи о поиске гамильтоновых циклов или, если граф «нагруженный», о поиске путей коммивояжера. При больших размерностях задачи (значение N) переборные схемы не работают — остаются только приближенные методы решения. Суть приближенных методов заключается в быстром поиске первого решения (первой оценки), а затем в попытках его улучшения. В методе локальной оптимизации в этих попытках просматриваются только соседние вершины графа (города) найденного пути.

Шаг 1. Получить приближенное решение (первую оценку).

Шаг 2. Пока происходит улучшение решения, выполнять следующий шаг, иначе перейти на шаг 4.

Шаг 3. Для всех пар номеров городов i,j, удовлетворяющих неравенству ($1 \le i < j \le N$), проверить: $d_{i-1,i}+d_{i,j}+d_{j,j+1} > d_{i-1,j}+d_{j,i}+d_{i,j+1}$ для смежных городов, т. е.
$j=i+1$, $d_{i-1,i}+d_{i,i+1}+d_{j-1,j}+d_{j,j+1} > d_{i-1,j}+d_{j,i+1}+d_{j-1,i}+d_{i,j+1}$
для несмежных городов.

Примечание

На рисунке 4.51 даны графические иллюстрации первого и второго неравенств. «Жирными» линиями обозначены участки старых маршрутов, «тонкими» — новых.

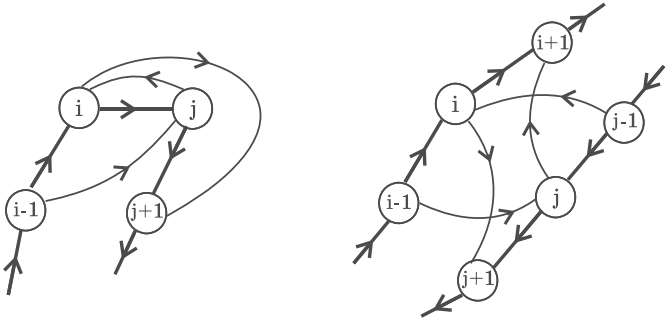


Рис. 4.51

Если одно из неравенств выполняется, то найдено лучшее решение. Следует откорректировать ранее найденное решение и вернуться на шаг 2.

Шаг 4. Закончить работу алгоритма.

Для реализации алгоритма достаточно матрицы расстояний A и массива для хранения пути коммивояжера Way.

Вид общей алгоритма:

```
Begin
 Init;{*Ввод из файла матрицы расстояний,
       инициализация глобальных переменных.*}
 OneWay;{*Поиск первого варианта пути
          коммивояжера.*}
 Local;{*Локальная оптимизация.*}
 Out;{*Вывод результата.*}
End.
```

Продолжим уточнение алгоритма.

Работа процедур *Init*, *OneWay*, *Out* достаточно очевидна. Естественным приемом является вынесение ее в самостоятельную часть работы школьников на занятии.

Нам необходимо уточнить процедуру *Local*. Работаем не с частностями, а на содержательном уровне. Предположим, что мы имеем функции *Best1* и *Best2*, их параметры — индексы элементов массива *Way*, определяющих номера городов в пути коммивояжера, а выход естественный — истина или ложь, в зависимости от того, выполняются неравенства или нет. Рассуждаем дальше. Если неравенство выполняется, то нам необходимо изменить путь (соответствующие элементы массива *Way*). Пока не будем заниматься деталями. Пусть эту работу выполняет процедура *Swap*, ее параметры — индексы элементов массива *Way*). Эта процедура, вероятно, должна сообщать о том, что она «что-то изменила», ибо нам необходимо продолжать работу до тех пор, пока что-то меняется, происходят улучшения.

Итак, алгоритм процедуры *Local*.

```
Procedure Local;
 Var i,j:Integer;
      Change:Boolean;
 <здесь функции Best1 и Best2, а также процедура
 Swap>;
 Begin
  Repeat
   Change:=False;
   For i:=1 To N-1 Do
    For j:=i+1 To N Do
     If i=j+1 Then
       Begin
         If Best1(i,j) Then Swap(i,j);
       End
                        Else
       If (i=1) And (j=N)
         Then
           Begin
             If Best1(i,j) Then Swap(i,j);
           End {*Об этой проверке лучше первоначально
                умолчать, чтобы было о чем спросить,
                "если совсем будет плохо" - все понимают
                и нет вопросов.*}
         Else
          If Best2(i,j) Then Swap(i,j);
 Until Not(Change);{*Из логики неясно - кто
       изменяет значение переменной Change на True.
       Определите, учитывая, что функции Best1,
       Best2 и процедура Swap локальные по
       отношению к процедуре Local.*}
End;
```

4.10.2. Алгоритм Эйлера

Этот алгоритм и следующий работоспособны в том случае, если выполняется неравенство треугольника. Его суть в том, что для любой тройки городов *i, j, k* (между которыми есть связь) выполняется неравенство $d_{i,j}+d_{j,k}>d_{i,k}$.

Рассмотрим идею алгоритма.

Шаг 1. Строится каркас минимального веса (алгоритмы Прима или Краскала).

Примечание. Это не есть первое приближение, как в предыдущем алгоритме.

Шаг 2. Путем дублирования каждого ребра каркас преобразуется в эйлеров граф.

Шаг 3. Находим в построенном графе эйлеров цикл.

Шаг 4. Эйлеров цикл преобразуем в гамильтонов цикл (или маршрут коммивояжера). Метод преобразования: последовательность вершин эйлерова цикла сокращается так, чтобы каждая вершина графа в получившемся цикле встречалась ровно один раз.

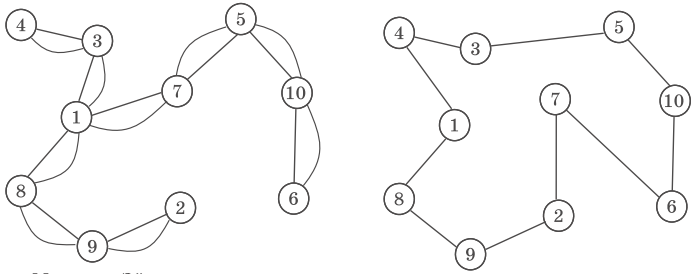
Шаг 5. Закончить работу алгоритма. Получено приближенное решение задачи коммивояжера.

Покажем, что стоимость приближенного решение *CostAp* не превосходит удвоенной стоимости оптимального решения *CostBet*.

Пусть стоимость каркаса — *CostFr*. Тогда *CostFr*<*CostBet*, так как при удалении из оптимального пути коммивояжера ребра получаем каркас с весом не большим, чем *CostAp*. Из правила построения эйлерова графа получаем, что вес построенного эйлерова цикла 2×*CostFr*. Неравенство треугольника обеспечивает результат: следующую оценку шага 4 — *CostAp*<2×*CostFr*, а значит, *CostAp*<2×*CostBet*.

Пример.

Рассмотрим пример на рисунке 4.52.



Маршрут Эйлера:
1-8-9-2-9-8-1-7-5-10-6-10-5-7-1-3-4-3-1

Рис. 4.52

Для исходных данных, приведенных в таблице 4.9, «жирным» шрифтом выделен каркас минимального веса. В таблице выделены те ребра графа, которые включаются в каркас с помощью алгоритма Прима.

Таблица 4.9

	1	2	3	4	5	6	7	8	9	10
1	@	32	***19***	33	22	41	***18***	***15***	16	31
2	32	@	51	58	27	42	35	18	***17***	34
3	***19***	51	@	***23***	35	49	26	34	35	41
4	33	58	***23***	@	33	37	23	46	46	32
5	22	27	35	33	@	19	***10***	23	23	***9***
6	41	42	49	37	19	@	24	42	42	***10***
7	***18***	35	26	23	***10***	24	@	25	25	14
8	***15***	18	34	46	23	42	25	@	***1***	32
9	16	***17***	35	46	23	42	25	***1***	@	32
10	31	34	41	32	***9***	***10***	14	32	32	@

Последовательность включения ребер в каркас: (8,9), (8,1), (9,2), (1,7), (7,5), (5,10), (10,6), (1,3) и (3,4).

«Тонкие» линии добавляются на шаге 2. Получаем эйлеров граф. Затем эйлеров цикл (его стоимость 244) преобразуется в путь коммивояжера (шаг 4). Согласно неравенству треугольника мы получаем:

$$d_{2,9}+d_{9,8}>d_{2,8};\ d_{2,8}+d_{8,1}>d_{2,1};\ d_{2,1}+d_{1,7}>d_{2,7}.$$

Суммируя левые и правые части неравенств, получаем:

$$d_{2,9}+d_{9,8}+d_{8,1}+d_{1,7}>d_{2,7}.$$

Аналогичные преобразования осуществляются и для других фрагментов сокращения эйлерова цикла. Для рассмотренного примера *CostAp* равна 202.

Оцените, насколько стоимость этого маршрута больше стоимости оптимального?

Структуры данных. Помимо названных ранее массивов *A* — матрица расстояний и массива *Way* — хранение искомого пути, требуется «что-то» для хранения описания каркаса. Пусть это будет массив *B*:

```
Array[1..Nmax,1..Nmax] Of Boolean.
```

Элемент *B*[*i*,*j*], равный *True*, говорит о том, что ребро (*i*,*j*) графа принадлежит каркасу.

Общий алгоритм.

```
Begin
 Init;{*Ввод описания - матрица расстояний;
       инициализация данных.*}
 Solve;{*Решение задачи.*}
 Out;{*Вывод результата.*}
End.
```

Процедуры *Init* и *Out* очевидны (должны создаваться «на автомате»). Уточняем процедуру *Solve*. Первый «набросок».

```
Procedure Solve;
 Var ?{*Пока не знаем.*}
 <процедуры и функции>;
 Begin
  InitSolve;{*Инициализация переменных процедуры
       Solve.*}
  FindTree;{*Построение каркаса.*}
  EilerWay;{*Поиск эйлерова цикла.*}
  KommWay;{*Поиск пути коммивояжера.*}
 End;
```

Прежде чем продолжать уточнение, необходимо определить структуры данных этой части алгоритма и взаимодействие его составных частей. Во-первых, при построении каркаса необходимо иметь список ребер, отсортированный в порядке возрастания их весов (алгоритмы Краскала и Прима). Следовательно, на входе процедуры *FindTree* должна быть матрица расстояний *A*, на выходе — *B*, рабочие структуры — массив для хранения списка ребер. Внутренние процедуры: формирование списка ребер и сортировки.

Продолжим рассмотрение. Эйлеров цикл необходимо где-то хранить. Пусть это будет массив *St* (*Array*[1..*N**(*N*−1) *Div* 2] *Of Byte*). Количество ненулевых элементов в *St* — значение переменной *Count*. Эти величины описываются в разделе переменных процедуры *Solve*, их инициализация — в процедуре *InitSolve*. Процедуру поиска эйлерова цикла сделаем рекурсивной, поэтому первый вызов изменится — *EilerWay*(1). Выбор начальной вершины при поиске эйлерова цикла не имеет значения.

Итак, классический алгоритм поиска эйлерова цикла приводится с целью показа работы процедуры *KommWay*, ибо последняя не есть поиск гамильтонова цикла в обычной трактовке.

```
Procedure EilerWay(v:Byte);
 Var j:Integer;
 Begin
  For j:=1 To N Do
   If B[v,j] Then
     Begin
       B[v,j]:=False; B[j,v]:=False;EilerWay(j);
     End;
  Inc(Count);St[Count]:=v;{*Заносим номер вершины
      в эйлеров цикл.*}
 End;
Procedure KommWay;
 Var s:Set Of 1..Nmax;
       i,j:Integer;
 Begin
  i:=0;s:=[];
  For j:=1 To Count Do{*Исключаем повторяющиеся
      номера вершин из эйлерова цикла.*}
   If Not(St[j] In s) Then
     Begin
        Inc(i);
        Way[i]:=St[j];
        s:=s+[St[j]];
     End;
 End;
```

4.10.3. Алгоритм Кристофидеса

Отличается от предыдущего алгоритма построением маршрута из каркаса минимального веса. Неравенство треугольника выполняется.

Шаг 1. Строится каркас минимального веса (алгоритм Прима или Краскала).

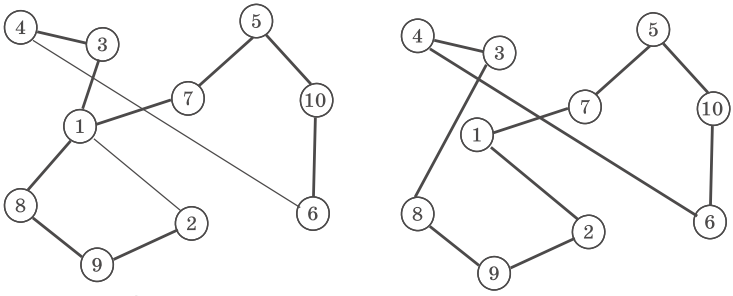
Шаг 2. На множестве вершин каркаса (обозначим их через V'), имеющих нечетное число ребер каркаса, находится паросочетание минимального веса. Таких вершин в любом каркасе четное число. Метод поиска паросочетания — чередующиеся цепочки.

Шаг 3. Каркас преобразуется в эйлеров граф путем присоединения ребер, соответствующих найденному паросочетанию.

Шаг 4. В построенном графе находится эйлеров маршрут.

Шаг 5. Эйлеров маршрут преобразуется в маршрут коммивояжера.

Итак, данный метод отличается от предыдущего только шагами 2 и 3. Все этапы реализуются за полиномиальное время. Известно [16], что для этого метода оценка приближенного метода имеет вид $CostAp<1.5\times CostBest$ и эта оценка является лучшей для приближенных полиномиальных алгоритмов решения задачи о коммивояжере с неравенством треугольника.



Маршрут Эйлера:
1-7-5-10-6-4-3-1-8-9-2-1

Рис. 4.53

На данных из предыдущего параграфа на рисунке 4.53 показан пример каркаса («жирными» линиями) и паросочетание минимального веса («тонкими» линиями), построенное на вершинах каркаса с нечетными степенями (вершинах 1, 2, 4, 6). Подграф имеет матрицу расстояний, приведенную в таблице 4.10.

Таблица 4.10

1	2	4	6
∞	32	33	41
32	∞	58	42
33	58	∞	37
41	42	37	∞

Путь коммивояжера имеет стоимость *CostAp*, равную 191.

В алгоритм *Solve* (см. предыдущий алгоритм) добавляется процедура построения паросочетания минимального веса (*P*). Назовем ее *Pair*. Ее входными данными является матрица *B* (описывает каркас), выходными — новая матрица *C* (элементы логического типа), соответствующая графу, получаемому добавлением к каркасу ребер *P*.

```
Procedure Pair;
 Var ?{*Пока не знаем.*};
 <процедуры и функции, вложенные в Pair>;
 Begin
  InitPair;{*Инициализация переменных процедуры,
      формирование массива с номерами вершин,
      имеющих нечетную степень.*}
  First;{*Поиск первого паросочетания.*}
  Find;{*Поиск P.*}
  Ad;{*Добавление ребер, образующих P, к каркасу -
      матрица C.*}
 End;
```

4.11. Задачи

1. Дан ориентированный граф с N вершинами ($N<50$). Вершины и дуги окрашены в цвета с номерами от 1 до M ($M\le 6$). Указаны две вершины, в которых находятся фишки игрока и конечная вершина. Правило перемещения фишек: игрок может передвигать фишку по дуге, если ее цвет совпадает с цветом вершины, в которой находится другая фишка; ходы можно делать только в направлении дуг графа; поочередность ходов необязательна. Игра заканчивается, если одна из фишек достигает конечной вершины.

Написать программу поиска кратчайшего пути до конечной вершины, если он существует.

2. Некоторые школы связаны компьютерной сетью. Между школами заключены соглашения: каждая школа имеет список школ-получателей, которым она рассылает программное обеспечение всякий раз, получив новое бесплатное программное обеспечение (извне сети или из другой школы). При этом, если школа B есть в списке получателей школы A, то школа A может не быть в списке получателей школы B.

Написать программу, определяющую минимальное количество школ, которым надо передать по экземпляру нового программного обеспечения, чтобы распространить его по всем школам сети в соответствии с соглашениями.

Кроме того, надо обеспечить возможность рассылки нового программного обеспечения из любой школы по всем остальным школам. Для этого можно расширять списки получателей некоторых школ, добавляя в них новые школы. Требуется найти минимальное суммарное количество расширений списков, при которых программное обеспечение из любой школы достигло бы всех остальных школ. Одно расширение означает добавле-

ние одной новой школы-получателя в список получателей одной из школ.

3. Задан неориентированный граф. При прохождении по некоторым ребрам отдельные (определенные заранее) ребра могут исчезать или появляться.

Написать программу поиска кратчайшего пути из вершины с номером q в вершину с номером w.

4. Заданы два числа N и M ($20 \le M \le N \le 150$), где N — количество точек на плоскости. Написать программу построения дерева из M точек так, чтобы оно было оптимальным.

Дерево называется оптимальным, если сумма всех его ребер минимальна. Все ребра — это расстояния между вершинами, заданными координатами точек на плоскости.

5. Даны два числа N и M. Написать программу построения графа из N вершин и M ребер. Каждой вершине ставится в соответствие число ребер, входящих в нее. Граф должен быть таким, чтобы сумма квадратов этих чисел была минимальна.

6. Задан ориентированный граф с N вершинами, каждому ребру которого приписан неотрицательный вес. Написать программу поиска простого цикла, для которого среднее геометрическое весов его ребер было бы минимально.

Примечание

Цикл называется простым, если через каждую вершину он проходит не более одного раза, петли в графе отсутствуют.

7. Компьютерная сеть* состоит из связанных между собой двусторонними каналами связи N компьютеров (N не превосходит 50, а количество каналов $N+5$), номера которых от 1 до N. Эта сеть предназначена для распространения сообщения от центрального компьютера всем остальным. Компьютер, получивший сообщение, владеет им и может распространять его дальше по сети. Запрещается передавать сообщения одному и тому

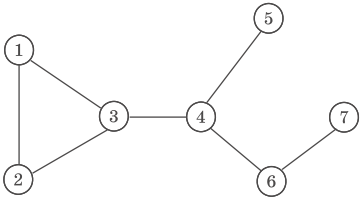


Рис. 4.54

* Задача предложена Антоном Александровичем Сухановым для Всероссийской олимпиады школьников в 1993 году.

же компьютеру дважды. Время передачи сообщения по каналу связи занимает одну секунду, при этом передающий и принимающий компьютеры заняты на все время передачи данного сообщения. На рисунке 4.54 приведен возможный вариант такой сети.

В начальный момент времени центральный компьютер может передать сообщение одному из непосредственно связанных с ним компьютеров, т. е. соседу. После окончания передачи этим сообщением будут владеть оба компьютера. Каждый из них может передать сообщение одному из своих соседей и так далее, пока все компьютеры не будут владеть сообщением. Для сети, показанной на рисунке, возможный порядок распространения сообщения от центрального компьютера с номером 6 имеет вид:

Секунда 1: 6→4
Секунда 2: 4→3
6→7
Секунда 3: 3→1
4→5
Секунда 4: 3→2

Написать программу определения и вывода порядка передачи сообщения за минимальное время.

8. Внутрь квадрата с координатами левого нижнего угла (0,0) и координатами верхнего правого угла (100, 100) поместили N ($1 \le N \le 30$) квадратиков. Написать программу поиска кратчайшего пути из точки (0,0) в точку (100,100), который бы не пересекал ни одного из этих квадратиков. Ограничения:

- длина стороны каждого квадратика равна 5;
- стороны квадратиков параллельны осям координат;
- координаты всех углов квадратиков — целочисленные;
- квадратики не имеют общих точек.

Указание. Строится граф из начальной, конечной точек и вершин квадратиков (ребра не должны пересекать квадраты). Затем ищется кратчайший путь, например, с помощью алгоритма Дейкстры.

9. Задан неориентированный граф с N вершинами, пронумерованными целыми числами от 1 до N. Написать программу, которая последовательно решает следующие задачи:

- выясняет количество компонент связности графа;
- находит и выдает все такие ребра, что удаление любого из них ведет к увеличению числа компонент связности;
- определяет, можно ли ориентировать все ребра графа таким образом, чтобы получившийся граф оказался сильно связным;

- ориентирует максимальное количество ребер, чтобы получившийся граф оказался сильно связным;
- определяет минимальное количество ребер, которые следует добавить в граф, чтобы ответ на третий пункт был утвердительным.

10. Задан ориентированный граф с N ($1 \le N \le 33$) вершинами, пронумерованными целыми числами от 1 до N. Напишите программу, которая подсчитывает количество различных путей между всеми парами вершин графа.

11. Ребенок нарисовал кружки и некоторые из них соединил отрезками. Кружки он пометил целыми числами от 1 до N ($1 \le N \le 30$), а на каждом отрезке поставил стрелочку. Затем он приписал каждому кружочку вес в виде некоторого целого числа и определил начальный и конечный кружочки. Из первого он должен выйти, а во второй попасть.

Ребенок решил для себя следующее:

- набрать максимально возможное суммарное количество очков;
- по каждому отрезку пройти ровно один раз;
- если в кружок он попадает при движении по направлению стрелки, то к суммарному количеству очков вес этого кружка прибавляется;
- если в кружок он попадает при движении против направления стрелки, то из суммарного количества очков вес этого кружка вычитается.

Написать программу, которая бы помогла ребенку построить путь, удовлетворяющий всем этим требованиям.

12. Имеется поисковая система, состоящая из броневика и нескольких разведчиков. Эта система была направлена для поисков сейфа с секретной документацией противника. В лабиринте (см. рисунок 4.55) один из посланных разведчиков обнаружил цель поисков — сейф. Сам разведчик вывезти сейф не в состоянии, но броневик, с которого были посланы разведчики, может это сделать. Разведчик составил план своего пути в виде списка команд U (вверх), D (вниз), L (влево), R (вправо). Путь разведчика не обязательно является кратчайшим, но он точно безопасен (в лабиринте есть мины). Броневик поедет по пути, проходящему по пути разведчика (причем из квадрата A в соседний квадрат B он может ехать только в том случае, если разведчик тоже проходил из A в B, а проход разведчика из B в A не обеспечивает безопасности) так, чтобы истратить как можно меньше топлива. Написать программу, определяющую, сколько топлива нужно броневику, чтобы доехать до сейфа.

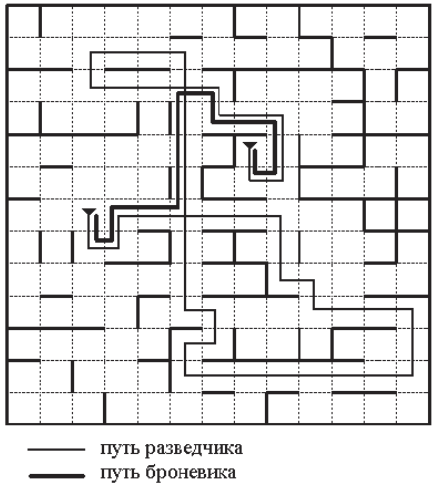


Рис. 4.55

Входные данные.

В первой строке входного файла записаны целые числа M и T (M — сколько топлива нужно на движение на 1, T — сколько топлива нужно на поворот на 90°). $1 \le M, T \le 100$.

Во второй строке записано целое число N ($0 \le N \le 100$).

В третьей — строка из N символов, задающая движение разведчика, каждый символ которой — это один из четырех символов *U, D, L, R*.

Выходные данные.

Выведите в выходной файл минимальное количество топлива, необходимое броневику для того, чтобы доехать до сейфа.

Примеры.

input.txt

2 1

51

DRURRRRRDDRDRRRDDLLLLLLLURULUUUUUUUULLL-DRRRRDRRDDLU

output.txt

44

input.txt

10 1

9

RULDRUDUL

output.txt

32

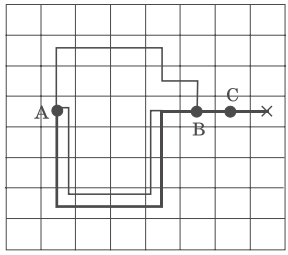


Рис. 4.56

Указание. Упростим задачу. Пусть нет стоимостей поворота. Так как путь описывается строкой не более чем из 100 символов, то по нему можно построить ориентированный граф, в котором будет максимум 100 вершин. Стоимость дуг известна, и она постоянна в этом случае. Осталось с помощью известного алгоритма Дейкстры (поиск пути с минимальным весом) найти решение. Что изменяется при введении стоимости поворота? Рассмотрим пример, приведенный на рисунке 4.56. Из точки A дошли до точки B, затем вернулись в точку A и другим путем снова пришли в точку B. Таким образом, из точки A в точку B можно попасть двумя способами. Верхний путь содержит 7 перемещений и 4 поворота, нижний — 9 перемещений и 3 поворота. Пусть стоимость перемещения равна 2, а стоимость поворота — 3. Тогда верхний путь имеет общую стоимость 26, а нижний — 27. Казалось бы, что следует выбирать верхний путь. Однако, если оценить стоимости перемещений до точки C, то окажется, что верхний путь имеет стоимость 30, а нижний — 28. Из этого следует, что для каждой вершины необходимо хранить не одну минимальную стоимость ее достижения, а четыре стоимости — по каждому направлению, строить соответствующий граф и применять алгоритм Дейкстры. В этом заключается «изюминка» задачи.

13. «*Химическая тревога*». Произошло радиоактивное заражение местности. Составлена карта зараженности. Она представляет собой прямоугольную таблицу $N \times M$, в клетках которой записана зараженность соответствующего участка. Написать программу поиска пути из левой верхней клетки таблицы в правую нижнюю клетку с минимальной суммарной дозой радиации.

Входные данные. В первой строке входного файла *input.txt* записаны числа N и M, а в следующих N строках — по M чисел — карта зараженности местности. Числа в строке разделяются одним или несколькими пробелами. $1 \le N \le 30$, $1 \le M \le 30$, зараженность участка — целое число от 0 до 100.

Выходные данные. В файл *output.txt* записывается одно число — суммарная доза радиации.

На рисунке 4.57 приведен пример заполнения входных и выходных данных.

Указание. В явном виде динамическую схему применить нельзя, ибо даны не два направления движения, как в классиче-

ских задачах о Черепашке, а четыре. Однако если клетки таблицы рассматривать как вершины графа с ребрами, означающими возможность перехода из одной клетки в другую, то задача сводится к поиску пути в графе между заданными вершинами с минимальной стоимостью. Так как стоимости положительные, то применим алгоритм Дейкстры. Данная схема в чистом виде неприменима. Например, при описании графа с использованием матрицы смежности потребуется использовать двумерный массив 900×900 байт. Оперативной памяти явно не хватит. Следует идти другим путем. Достаточно ограничиться массивом оценок достижения каждой клетки таблицы (*Mn*). На каждом шаге обработки находим элемент *Mn* с минимальной оценкой. Согласно алгоритму Дейкстры эта оценка окончательна. Вычисляем стоимости достижения клеток, соседних с найденной, помечаем клетку как обработанную (знак минус у значения оценки стоимости достижения данной клетки), и находим следующий элемент из *Mn*. Для примера на рисунке 4.56 окончательный вид массива *Mn* представлен в таблице 4.11. Ответ задачи — значение *Abs(Mn[N,M])*.

Пример.

input.txt

```
3 5
2 100   0 100 100
1 100   0   0   0
1   0   3 100   2
```

output.txt

```
9
```

Рис. 4.57

Таблица 4.11

	1	2	3	4	5
1	–2	–102	–7	–107	–7
2	–3	–103	–7	–7	–7
3	–4	–4	–7	–107	–9

14. Дана доска 8×8. На ней размещены один король и N коней ($0 \le N \le 63$). Фигуры перемещаются по шахматным правилам. В процессе перемещений в одной клетке разрешается размещать несколько фигур. Цель перемещений — собрать все фигуры в одной клетке, причем за минимальное общее количество перемещений. Дополнительное условие — если король и конь находятся в одной клетке, то их можно переместить вместе по правилу хода коня, причем это перемещение засчитывается за один ход.

Входные данные. Файл *input.txt* содержит одну строку символов без пробелов, описывающую начальное расположение фигур на доске. Строка содержит последовательность клеток

доски, первая из которых — клетка короля, остальные — клетки коней. Каждая клетка описывается парой буква-цифра.

Выходные данные. Файл *output.txt* должен содержать одно число — минимальное количество перемещений, за которое все фигуры можно собрать в одной клетке доски.

На рисунке 4.58 приведен пример заполнения входных и выходных данных.

Пример.
input.txt
D4A3A8H1H8
output.txt
10

Рис. 4.58

Указание. Представим доску в виде графа (см. рисунок 4.59). Клетки (вершины) соединены ребром, если из одной в другую можно попасть ходом коня (матрица смежности графа). Расстояние между двумя вершинами — это количество ходов, которые требуется сделать коню, чтобы попасть из одной клетки в другую. С помощью стандартного алгоритма находим кратчайшие пути между всеми парами вершин (алгоритм Флойда). Предположим, что пока решается задача только для коней. Для ответа на вопрос задачи необходимо найти клетку, до которой расстояние от всех клеток с конями минимально. При известных кратчайших расстояниях это поиск минимального значения по массиву. Количество ходов для короля из клетки (i, j) в (p, z) равно $Max(Abs(i\text{-}p), Abs(j\text{-}z))$.

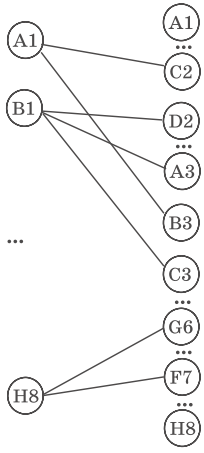


Рис. 4.59

Рассмотрим случай, когда конь может «подвозить» короля. Пусть клетка (i,j) — место встречи фигур и sc — оценка количества ходов коней до этой клетки. Необходимо найти коня и клетку доски, в которой он подхватит короля, причем вклад в общую оценку ходов при этом должен быть минимальным.

15. Задан ориентированный граф с N вершинами, пронумерованными целыми числами от 1 до N. Написать программу, которая подсчитывает количество различных путей между всеми парами вершин графа.

Входные данные. Входной файл *input.txt* содержит количество вершин графа N ($1 \le N \le 33$) и список дуг графа, заданных номерами начальной и конечной вершин.

Выходные данные. Вывести в выходной файл *output.txt* матрицу $N \times N$, элемент (i,j) которой равен числу различных путей, ведущих из вершины i в вершину j или -1, если существует бесконечно много таких путей.

Пример

На рисунке 4.60 приведен пример заполнения входных и выходных данных.

input.txt
5
1 2
2 4
3 4
4 1
5 3
1 1

output.txt
-1 -1 0 -1 0
-1 -1 0 -1 0
-1 -1 0 -1 0
-1 -1 0 -1 0
-1 -1 1 -1 0

Рис. 4.60

Указание. В алгоритмах на графах есть очень изящный метод определения кратчайших путей между всеми парами вершин — алгоритм Флойда. Используем его идею для решения нашей задачи. Пусть в матрице $F[1..N,1..N]$ в текущий момент времени хранится количество путей длины r между всеми парами вершин, и нам необходимо найти количество путей длины $r+1$. Что делать? При $F[i,k]<>0$ между вершинами i и k есть какое-то количество путей длины r и если при этом $A[k,j]<>0$, то между вершинами i и j есть пути длины $r+1$. Просчитывая для всех значений k, мы получаем количество путей между всеми парами вершин длины $r+1$. Второе соображение касается циклов в графе, ибо они определяют значение бесконечности (число путей) задачи. Очевидно, что если выявлен цикл, то до всех вершин этого цикла существует бесконечно много путей. Выявленный цикл — это ненулевой элемент на главной диагонали матрицы F при какой-либо из N итераций. Почему N? Да очень просто, N — это длина максимального цикла в графе.

16. Дано прямоугольное клетчатое поле $M \times N$ клеток. Каждая клетка поля покрашена в один из шести цветов, причем верхняя левая и нижняя правая имеют различный цвет. В результате поле разбивается на некоторое количество одноцветных областей: две клетки одного цвета, имеющие общую сторону, принадлежат одной области.

Правила игры. За первым игроком закреплена область, включающая левую верхнюю клетку, за вторым — правую нижнюю. Игроки ходят по очереди. Делая ход, игрок перекрашивает свою область по своему выбору:

- в любой из шести цветов;
- в любой из шести цветов, за исключением цвета своей области и цвета области противника.

В результате хода к области игрока присоединяются все прилегающие к ней области выбранного цвета, если такие имеются. Если после очередного хода окажется, что области игроков соприкасаются, то игра заканчивается.

Написать программу, которая определяет минимально возможное число ходов, по прошествии которых игра может завершиться.

Входные данные. Цвета пронумерованы цифрами от 1 до 6. Первая строка входного файла *input.txt* содержит M и N — размеры поля ($1 \le M, N \le 50$). Далее следует описание раскраски поля — M строк по N цифр (от 1 до 6) в каждой без пробелов. Первая цифра файла соответствует цвету левой верхней клетки игрового поля. Количество одноцветных областей не превосходит 50.

Выходные данные. В выходной файл *output.txt* необходимо вывести искомое количество ходов для каждого из пунктов.

На рисунке 4.60 приведен пример заполнения входных и выходных данных.

Пример

input.txt

```
4 3
1 2 2
2 2 1
1 4 3
1 3 2
```

output.txt

```
3
4
```

Рис. 4.61

Указание. Сведем задачу к некоторой графовой модели. Рассмотрим идею решения на примере, приведенном в формулировке задачи. Преобразуем исходную матрицу (T) (см. рисунок 4.61) в матрицу областей (Ob) (см. рисунок 4.62), определим цвет каждой области (массив C) и подсчитаем их количество (k). Для примера Ob будет выглядеть следующим образом. Массив C — (1, 2, 1, 1, 4, 3, 3, 2), k равно 8. А затем по этой матрице построим матрицу смежности графа (G), определяющего связи между областями (см. рисунок 4.63).

```
1 2 2
2 2 3
4 5 6
4 7 8
```

Рис. 4.62

После этих замечаний решение первого пункта задачи сводится к поиску кратчайшего пути, точнее количества ребер в пути, между вершинами 1 и 8 (см. рисунок 4.64). Для этого можно использовать алгоритм Дейкстры или обход графа в ширину. Ответ задачи — значение уровня вершины 8 (при обходе в ширину) минус единица.

```
0 1 0 0 0 0 0 0
1 0 1 1 1 0 0 0
0 1 0 0 0 1 0 0
0 1 0 0 1 0 1 0
0 1 0 1 0 1 1 0
0 0 1 0 1 0 0 1
0 0 0 1 1 0 0 1
0 0 0 0 0 1 1 0
```

Рис. 4.63

Чем отличается решение по второму пункту задачи? Нельзя окрашивать область в цвета своей области и области противника. Это приводит к тому, что мы можем остаться в той же вершине графа областей. Опишем состояние игры кортежем (x,y,*play*,*color*), где x —

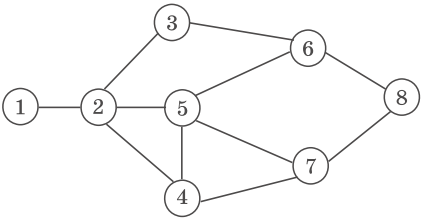


Рис. 4.64

номер области первого игрока, *y* — номер области второго игрока, *play* — номер игрока и *color* — номер цвета, в который игрок окрашивает область. Как выполнить такой просмотр состояний игры? Да опять же традиционным обходом в ширину в пространстве состояний. Завершением обхода будет или просмотр всей очереди, или достижение одного из конечных состояний. Принцип обхода в ширину дает минимальное возможное число ходов, по прошествии которых игра завершается.

17. Корпоративная сеть состоит из N компьютеров, некоторые из них соединены прямыми двусторонними каналами связи. В целях повышения секретности при проектировании сети количество каналов связи было сокращено до минимума с тем условием, чтобы любые два компьютера имели возможность обмена информацией либо непосредственно, либо через другие компьютеры сети.

Хакер хочет прослушивать все передаваемые в сети сообщения. Для этого он разработал вирус, который, будучи установленным на какой-либо из компьютеров, передает ему всю информацию, проходящую через этот компьютер. Оказалось, что материальные затраты, необходимые для установки вируса на конкретный компьютер, различны.

Требуется определить набор компьютеров, которые хакер должен заразить вирусом, чтобы минимизировать общие материальные затраты.

Входные данные. Первая строка входного файла *input.txt* содержит N — количество компьютеров в сети ($1 \le N \le 500$). В i-й из последующих N строк содержатся номера компьютеров, с которыми непосредственно связан компьютер с номером i. Далее следует N целых чисел из диапазона [1,1000] — материальные затраты, связанные с установкой вируса на каждый из компьютеров.

Выходные данные. В выходной файл *output.txt* требуется вывести минимально возможные суммарные затраты и список но-

меров компьютеров, которые необходимо инфицировать, упорядоченный по возрастанию.

На рисунке 4.66 приведен пример заполнения входных и выходных данных.

Пример

input.txt

```
5
5
4
4
2 3 5
4 1
1 5 5 2 10
```

output.txt

```
3
1 4
```

Рис. 4.65

Указание. В условии задачи в «сверхзавуалированной» форме сказано о том, что связи компьютеров описываются некоторой древовидной структурой, ибо в противном случае «сокращение до минимума» каналов связи невозможно. Итак, мы имеем какое-то дерево (связное), описывающее связи между компьютерами. Это ограничение дает возможность решать задачу и для 500 компьютеров. При связях компьютеров, описываемых произвольным связным графом, вероятно, ничего другого, кроме перебора с некоторыми эвристиками, не остается делать. А это уже достаточно безнадежно.

Идея решения. Рассматриваем вершины дерева по принципу обхода в ширину. Мы находимся в вершине с номером k (см. рисунок 4.66), и нам требуется оценить (с учетом требования задачи — все связи между компьютерами должны быть под контролем):

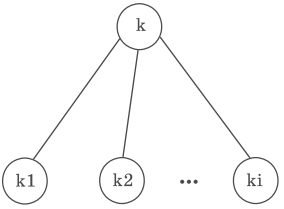


Рис. 4.66

- $B[k]$ — стоимость заражения компьютеров из поддерева, при условии, что и компьютер с номером k заражается;
- $W[k]$ — то же самое, только компьютер с номером k не заражается.

Так как используется обход в ширину, то оценки B и W для всех компьютеров с номерами k_1, k_2, ..., k_i уже получены. В первом случае — при оценке $B[k]$ — компьютеры k_1, k_2, ..., k_i мы можем как заражать, так и не заражать, т. е. необходимо найти

$$B[k]+min(B[k_1],W[k_1])+min(B[k_2],W[k_2])+\ldots+min(B[k_i],W[k_i]).$$

Во втором случае — не заражаем компьютер k — компьютеры k_1, k_2, ..., k_i необходимо заражать, в противном будет нарушено условие задачи, т. е. $W[k]=B[k_1]+B[k_2]+\ldots+B[k_i]$. Ответом на первый пункт задачи будет значение $min(B[j],W[j])$, где j — номер вершины, с которой начинается обход в ширину.

18. Дан ориентированный граф. Вершинам приписаны веса. Требуется найти путь между начальной и конечной вершинами (они заданы) с максимально возможным суммарным весом. Путь должен удовлетворять следующим условиям:

- по каждой дуге разрешается пройти один раз;
- если в вершину попадаем по входящей дуге, то к суммарному весу вес этой вершины прибавляется;
- если в вершину попадаем по выходящей дуге, то из суммарного веса вес этой вершины вычитается.

Входные данные. Входной файл *input.txt* содержит данные в следующей последовательности:

- N x_1 x_2 ... x_N — количество вершин графа и их веса ($1 \le N \le 30$, $1 \le x_i \le 30000$);
- b, e — номера начальной и конечной вершин;
- M — количество дуг;
- i_1 j_1 — номера вершин, соединяемых первой дугой;
- i_2 j_2 — номера вершин, соединяемых второй дугой;
- ...
- i_M j_M — номера вершин, соединяемых дугой с номером M.

Выходные данные. В выходной файл *output.txt* требуется вывести максимальный вес пути и путь, на котором он достигается. В случае, если решение не существует, выходной файл должен содержать единственную строку *«no solution»*.

Пример

input.txt
```
5 1 3 5 100
2 3
1 4
5
1 2
2 3
5 3
2 5
4 2
```

output.txt
```
-72
1 2 5 3 2 4
```

Рис. 4.67

На рисунке 4.67 приведен пример заполнения входных и выходных данных, а на рисунке 4.68 — граф, соответствующий входным данным.

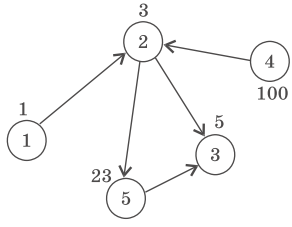


Рис. 4.68

Указание. Задача на поиск эйлерова пути в графе между заданными вершинами. Так как требуется найти не просто путь, а путь с наилучшей оценкой, то перебор вариантов, — это, наверное, единственный способ решения задачи. Эйлеров путь между вершинами b и e существует (теорема Эйлера) в связном графе в том и только том случае, если вершины b и e имеют нечетные степени, а все остальные — четные степени. Следовательно, на стадии предварительной обработки требуется выяснить существует ли в принципе путь между заданными вершинами.

Для сокращения перебора разумно ввести дополнительную структуру данных, в которой для каждой вершины определе-

на последовательность просмотра остальных вершин графа. Для нашего примера она может иметь вид, приведенный на рисунке 4.69.

2	0	0	0	0
5	3	1	4	0
2	5	0	0	0
2	0	0	0	0
3	2	0	0	0

Рис. 4.69

Так, из вершины 2 очередность просмотра вершин — 5, 3, 1, 4. Вероятность того, что при этом на первых шагах перебора получается оценка, близкая к оптимальной, возрастает, если не рассматривать специально подобранные тесты. Кроме того, следует ввести оперативную проверку связности непросмотренной части графа. Пусть мы идем по ветке перебора 1-4-6, как показано на рисунке 4.71. В процессе поиска решения — эйлерова пути — пройденные ребра «выбрасываются». Перечеркнутые ребра удалены, они пройдены, и мы находимся в 6-й вершине. Может оказаться, во-первых, что появятся изолированные вершины и, во-вторых, что нарушится связность оставшейся части графа. Например, так, как это показано на рисунке 4.70. Необходимо попасть из 1-й вершины в 7-ю. Очевидно, что вершины 2, 3, 4 уже никак не попадут в путь, поэтому дальнейший поиск пути в этом варианте решения не имеет смысла. Поиск эвристик, сокращающих перебор, можно продолжить.

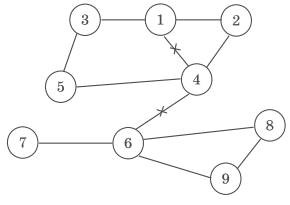


Рис. 4.70

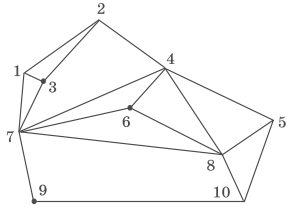


Рис. 4.71

19. «Стены»* . В некоторой стране стены построены таким образом, что каждая стена соединяет ровно два города, и стены не пересекают друг друга. Таким образом, страна разделена на отдельные области. Чтобы добраться

* Задача с Международной олимпиады школьников по информатике 2000 года.

из одной области в другую, необходимо либо пройти через город, либо пересечь стену. Два любых города A и B могут соединяться не более чем одной стеной (один конец стены в городе A, а другой — в городе B). Более того, всегда существует путь из города A в город B, проходящий вдоль каких-либо стен и через города.

Существует клуб, члены которого живут в городах. В каждом городе может жить либо один член клуба, либо вообще ни одного. Иногда у членов клуба возникает желание встретиться внутри одной из областей, но не в городе. Чтобы попасть в нужную область, каждый член клуба должен пересечь какое-то количество стен, возможно равное нулю. Члены клуба путешествуют на велосипедах. Они не хотят въезжать ни в один город из-за интенсивного движения на городских улицах. Они также хотят пересечь минимально возможное количество стен, так как пересекать стену с велосипедом довольно трудно. Исходя из этого, нужно найти такую оптимальную область, чтобы суммарное количество стен, которые требуется пересечь членам клуба, называемое суммой пересечений, было минимальным из всех возможных.

Города пронумерованы целыми числами от 1 до N , где N — количество городов. На рисунке 4.71 пронумерованные вершины обозначают города, а линии, соединяющие вершины, обозначают стены.

Предположим, что членами клуба являются три человека, которые живут в городах с номерами 3, 6 и 9. Тогда оптимальная область и соответствующие маршруты движения членов клуба показаны на рисунке 4.72. Сумма пересечений равняется 2, так как членам клуба следует пересечь две стены: человеку из города 9 требуется пересечь стену между городами 2 и 4, человеку из города 6 требуется пересечь стену между городами 4 и 7, а человек из города 3 не пересекает стен вообще.

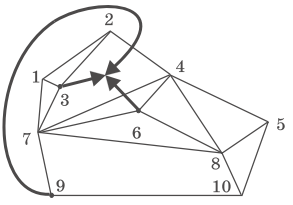


Рис. 4.72

Требуется написать программу, которая по заданной информации о городах, областях и местах проживания членов клуба находит оптимальную область и минимальную сумму пересечений.

Входные данные. Файл имеет имя *walls.in*. Первая строка содержит одно целое число: количество областей M, $2 \le M \le 200$. Вторая строка содержит одно целое число: количество городов N, $3 \le N \le 250$. Третья строка содержит одно целое число: количе-

ство членов клуба L, $1 \le L \le 30$, $L \le N$. Четвертая строка содержит L различных целых чисел в возрастающем порядке: номера городов, где живут члены клуба.

Оставшаяся часть файла содержит $2M$ строк, по паре строк для каждой из M областей. Первые две строки из них описывают первую область, вторые две строки — вторую область, и т. д. В каждой паре первая строка содержит количество городов I на границе области; вторая строка содержит номера этих I городов в порядке обхода границы области по часовой стрелке. Единственным исключением является последняя область — это находящаяся снаружи (внешняя) область, окружающая все города и другие области, и для нее порядок следования городов на границе задается против часовой стрелки. Порядок описания областей во входном файле задает целые номера этим областям. Область, описанная первой во входном файле, имеет номер 1, описанная второй — номер 2, и т. д. Обратите внимание, что входной файл содержит описание всех областей, образованных городами и стенами, включая находящуюся снаружи (внешнюю) область.

Выходные данные. Файл имеет имя *walls.out*. Первая строка этого файла содержит одно целое число: минимальную сумму пересечений. Вторая строка содержит одно целое число: номер оптимальной области. Если существует несколько различных решений, вам необходимо выдать лишь одно из них.

Пример

walls.in

```
10
10
3
3 6 9
3
1 2 3
3
1 3 7
4
2 4 7 3
3
4 6 7
3
4 8 6
3
6 8 7
3
4 5 8
4
7 8 10 9
3
5 10 8
7
7 9 10 5 4
2 1
```

walls.out

```
2
3
```

Рис. 4.73

На рисунке 4.73 приведен пример заполнения входных и выходных данных, а на рисунке 4.74 — граф, соответствующий входным данным.

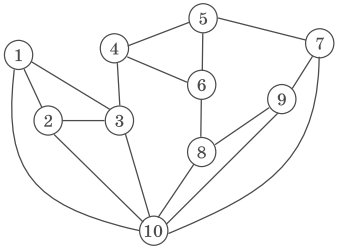


Рис. 4.74

Указание. В формулировке задачи содержится подсказка решения. Описание исходного графа, а это явная задача на графовые алгоритмы, дано, например, не в виде матрицы смежности, а заданием областей. Требуется построить граф, вершинами которого являются области, а ребрами — стены между областями. Естественно, что должна быть вершина и для внешней области. После формирования графа областей находим кратчайшие расстояния между всеми парами вершин (алгоритм Флойда). Для каждого члена клуба и каждой области определяем минимальное расстояние, за которое он добирается до этой области. Осталось просуммировать эти расстояния по всем членам клуба и выбрать минимум.

20. Два графа $G_a=(V_a,E_a)$ и $G_b=(V_b,E_b)$ называются изоморфными, если существует взаимно однозначное соответствие $f:V_a\to V_b$, такое, что сохраняется отношение смежности между вершинами графа, т. е. $(v,w)\in E_a$ тогда и только тогда, когда $(f(v),f(w))\in E_b$. Даны два графа. Определить, изоморфны ли они. На рисунке 4.75 приведены примеры изоморфных графов.

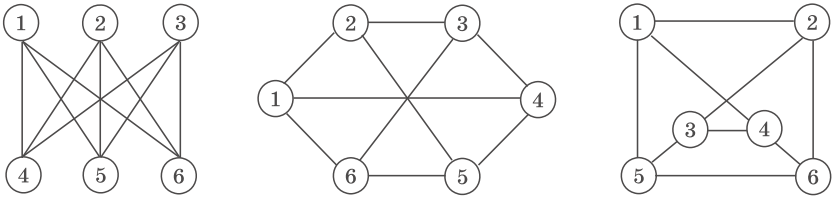


Рис. 4.75

Рекомендации. Прямой метод решения, заключающийся в генерации всех возможных перестановок $N!$, где N — мощность множеств V_a и V_b, и проверке сохранения отношения смежности для каждого случая работоспособен для небольших значений N ($N\le 12$). Несколько сократить перебор позволяет введение массивов для хранения степеней вершин (число ребер, инцидентных вершине) и установка соответствия только между вершинами, имеющими одну степень. Для графов, приведенных на рисунке 4.75 и имеющих одно значение степени, это сокращение не работает.

21. Максимально полный подграф графа G называется кликой. В клике между каждой парой вершин существует ребро. Число клик в графе может расти экспоненциально относительно числа вершин. Так, в графах M_N Муна–Мозера с $3\times N$ вер-

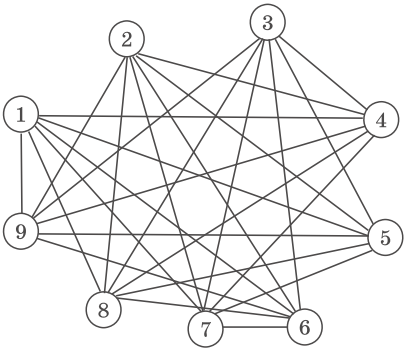


Рис. 4.76

шинами вершины разбиты на триады (1, 2, 3), (4, 5, 6), ... ($3\times N-2$, $3\times N-1$, $3\times N$). Ребер между вершинами любой триады нет. Однако вершины триады связаны ребрами со всеми вершинами, не принадлежащими данной триаде. На рисунке 4.75 приведен граф Муна–Мозера при $N=2$ (первый на рисунке), на рисунке 4.76 — при $N=3$.

Графы Муна — Мозера имеют 3^N клик, каждая из которых содержит N вершин. Перечислить для небольших значений N клики в графах Муна–Мозера.

Примечание

В общем случае задача о поиске клик в графе G решается достаточно сложно. Однако понятие клики противоположно понятию максимально независимого множества. Строим дополнение графа G'. Находим в G' максимально независимые множества (это мы умеем делать) — они соответствуют кликам исходного графа.

22. Рассмотрим четыре множества S_1=[2, 3], S_2=[1, 2, 4], S_3=[3, 4], S_4=[1, 3, 4]. Требуется выбрать такие различные четыре числа x_1, x_2, x_3, x_4, что $x_i \in S_i$, i=1, 2, 3, 4. Этими числами могут быть x_1=2, x_2=4, x_3=3, x_4=1, они представляют данную систему множеств. Несложно привести пример множеств, для которых данное представление невозможно. Ф. Холлом доказана теорема о существовании системы различных представителей. Система $M(S)=\{S_1, S_2, ..., S_N\}$ имеет систему различных представителей тогда и только тогда, когда для любой подсистемы $\{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\} \subseteq M(S)$ выполняется неравенство

$\left|\bigcup_{j=1}^{k} S_{i_j}\right| \geq k$, т. е. количество элементов в объединении любых k подмножеств должно быть не менее k.

Пусть S= [1, 2, ..., N]. Определить, имеет ли $M(S)$={S_1, S_2, ..., S_N} систему различных представителей.

Указание. По $M(S)$ требуется построить двудольный граф. Система различных представителей существует тогда и только тогда, когда в двудольном графе есть максимальное паросочетание из N ребер.

23. Дан массив $N\times M$, в клетках которого произвольным образом поставлены числа от 1 до $N\times M$. Горизонтальным ходом называется такая перестановка любого числа элементов массива, при которой каждый элемент остается в той строке, в которой он был. Вертикальным ходом называется такая перестановка любого числа элементов массива, при которой каждый элемент остается в том столбце, в котором он был. Разрешаются только горизонтальные и вертикальные ходы*.

Разработать программу, которая за возможно меньшее количество ходов расставляет элементы массива по порядку, т. е. в первой строке — от 1 до M, во второй строке — от M+1 до $2M$, и т. д.

Указание. Рассмотрим метод поиска системы различных представителей для нашей задачи на примере представленном на рисунке 4.77.

S_1	6	15	10	8
S_2	5	13	1	11
S_3	2	9	12	14
S_4	7	3	16	4

6	8	8	10	15	10
1	11	11	5	5	13
9	2	14	2	12	2
16	-	3	-	4	7

Рис. 4.77

Обозначим элементы каждой строки нашей таблицы как S_1, S_2, S_3, S_4. Считаем, что есть множества S_i ($1\le i\le N$), составленные из элементов, которые необходимо разбить на N классов K_i по принадлежности к строке результирующей таблицы. Эти множества удовлетворяют теореме Холла, то есть любые q множеств содержат элементы не менее чем q классов.

Начинаем работу.

Первая итерация (первый столбец правой таблицы). Из S_1 выбираем $6\in K_2$, из S_2 — $1\in K_1$, из S_3 — $9\in K_3$, так как 2 выбрать нельзя, класс K_1 занят, из S_4 — 16, так как 3, 4 и 7 выбирать не имеем права по той же причине.

---

* По мнению автора, это — одна из лучших олимпиадных задач Сергея Геннадьевича Волченкова.

Вторая итерация. По этому же алгоритму из S_1 — 8, S_2 — 11, S_3 — 2. Выбрать из S_4 нечего — все классы заняты. Строим чередующуюся цепочку (построение паросочетания в двудольном графе), она начинается в S_4 и состоит из ребер (S_4,3), (3,S_3), (S_3,14). Эти ребра на рисунке 4.78, помечены одинарными стрелками. Меняем представителя у S_3 на 14 из K_4, и для S_4 появляется представитель — 3 из K_1 (третий столбец правой таблицы).

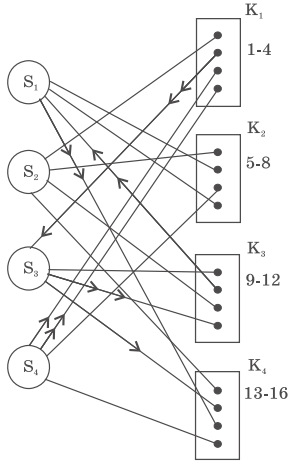


Рис. 4.78

Третья итерация. Выбираем для S_1 — 10, для S_2 — 5, для S_3 — 2. Проблемы выбора для S_4. Строим чередующуюся цепочку: (S_4,4), (2,S_3), (S_3,12), (10,S_1), (S_1,15), она выделена на рисунке двойными стрелками и меняем представителей (пятый столбец правой таблицы).

Четвертая итерация. Выбор представителей однозначен. Второй вертикальный и третий горизонтальный ходы очевидны. При вертикальном — каждый элемент столбца перемещаем в свою строку, а это можно сделать — у нас по одному представителю; при горизонтальном — каждый элемент строки перемещается в свой столбец.

24. Дан взвешенный (ребрам приписаны веса) граф G=(V,E), естественно связный.

Обозначим через $D(v,u)$ минимальное расстояние между вершинами $v,u \in V$.

Величина $D(G) = \underset{v,u \in V}{Max}(D(v,u))$ называется диаметром графа.

Величина $R(v) = \underset{u \in V}{Max}(D(v,u))$ максимальным удалением в графе от вершины v.

Величину $R(G) = \underset{v \in V}{Min}(R(v))$ называют радиусом графа.

Любая вершина $t \in V$, такая, что $R(t)=R(G)$, называется центром графа.

Написать программу поиска диаметра, радиуса и центров графа.

Указание. Первый шаг решения заключается в формировании матрицы минимальных расстояний между всеми парами вершин графа. Дальнейшее очевидно.

25. Плоским графом называется граф, вершины которого являются точками плоскости, а ребра — плоскими непрерывными линиями без самопересечений, соединяющими соответствующие вершины и не имеющими общих точек, естественно, кроме инцидентных им обеим вершинам. Граф, допускающий изображение в виде плоского, называют планарным. Известна теорема Понтрягина–Куратовского: «Граф планарен тогда и только тогда, когда он не содержит в качестве частей графы K_5 и $K_{3,3}$ (может быть с добавочными вершинами степени 2)». Графы K_5 и $K_{3,3}$ приведены на рисунке 4.79.

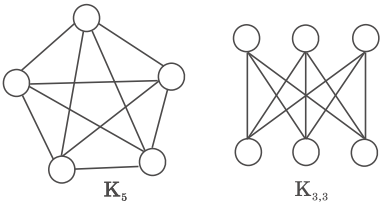


Рис. 4.79

Планарных графов достаточно мало при больших значениях N (количества вершин), и выделение в G подграфов типа K_5 и $K_{3,3}$ — достаточно сложная задача. Определить, какие подграфы содержатся в так называемом графе Петерсона, приведенном на рисунке 4.80.

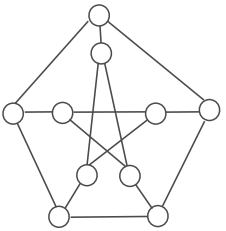


Рис. 4.80

26. Гранью (S) называют часть плоскости, окруженную простым циклом (циклом без самопересечений) и не содержащую внутри себя других элементов графа. Формула Л. Эйлера (1758 г.) связывает число граней, число вершин и ребер планарного графа $G=(V,E)$, где $|V|=N, |E|=M$: $S+N=M+2$. Если граф G без петель и кратных ребер (а в нашем рассмотрении участвуют именно такие графы), то каждая грань ограничена по крайней мере тремя ребрами графа. Из этого факта получается оценка снизу удвоенного числа ребер графа $3\times S\le 2\times M$. Отсюда следует оценка количества ребер планарного G графа — $M\le 3\times N-6$ (при $N\ge 3$). Проверить то, что графы K_5 и $K_{3,3}$ не являются планарными.

27. Согласно теореме Д. Кенига (1936 г.) для двудольности графа необходимо и достаточно, чтобы он не содержал циклов нечетной длины. Определить, является ли граф двудольным.

Указание. При поиске в ширину вершины графа помечаются метками 0, 1, 2 и т. д. Первой вершине, с которой начинается просмотр, приписывается значение 0, вершинам, связанным с ней, — метка 1 и т. д. Разобъем граф после просмотра на две части: X включает вершины с четными метками, Y — с нечетными. Если оба подграфа пусты — исходный граф является двудольным.

28. Задача Штейнера на графах. В связном взвешенном графе G с выделенным подмножеством вершин U ($U \subseteq V$) требуется найти связный подграф T, удовлетворяющий двум условиям:

- множество вершин подграфа T содержит заданное подмножество U;
- граф T имеет минимальный вес среди всех подграфов, удовлетворяющих первому условию.

Указание. Очевидно, что искомый подграф является деревом, оно называется деревом Штейнера. Задача сводится к нахождению дерева минимального веса в подграфах графа G, множество вершин которых содержит U. Эффективных алгоритмов решения задачи неизвестно, поэтому ничего другого не остается, как перебрать варианты при небольших значениях N.

29. Задача о пяти ферзях. На шахматной доске 8×8 расставить наименьшее число ферзей так, чтобы каждая клетка доски была под боем.

Указание. Построить граф для данной доски с учетом правил перемещения ферзя. Всякой искомой расстановке соответствует наименьшее доминирующее множество в графе.

30. Подмножество вершин графа, являющееся как независимым, так и доминирующим, называется ядром. Найти ядра графа.

Указание. Независимое множество является максимальным (не обязательно наибольшим) тогда и только тогда, когда оно является доминирующим. Таким образом, ядра графа — это независимые максимальные множества вершин.

5. Алгоритмы вычислительной геометрии

В этой главе рассматриваются алгоритмы, связанные с геометрией. Они служат хорошим «подспорьем» в изучении данного раздела математики. Но это не главное. Главное в том, что вычислительная геометрия (определение Ф. Препарата, В. Шеймоса [24]) является, во-первых, одним из разделов информатики и, во-вторых, *источником большого количества научных и прикладных задач*. Эта обширная область знаний, естественно, не рассматривается полностью, а только ее отдельные аспекты, доступные школьникам. Дело в том, что задач по информатике, связанных с вычислительной геометрией, достаточно много, а соответствующих публикаций достаточно мало. В какой-то степени материал этой главы восполняет данный пробел. Кроме того, специфика задач по вычислительной геометрии такова, что их реализация позволяет формировать у школьников структурный стиль написания программ.

5.1. Базовые процедуры

Прежде чем перейти к основным алгоритмам, рассмотрим типы данных и базовые процедуры работы с ними.

Точка на плоскости описывается парой вещественных чисел.

При использовании вещественного типа операции сравнения лучше оформить специальными функциями. Причина известна, на вещественных типах данных в системе программирования Паскаль нет отношения порядка, поэтому записи вида $a=b$, где a и b вещественные числа, лучше не использовать.

Приведем пример, объясняющий это положение.

Пусть для хранения вещественного типа используются два десятичных разряда для порядка и шесть разрядов для хранения мантиссы. Есть два числа:

$a=0{,}34567\times10^{4}$ и $b=0{,}98765\times10^{-4}$.

При сложении и вычитании осуществляется: выравнивание порядков, сложение (вычитание) мантисс и нормализация результата. После выравнивания порядков число $b=0{,}0000000098765\times10^{4}$. После сложения мантисс имеем: $0{,}3456700098765\times10^{4}$.

Так как для хранения мантиссы у нас шесть десятичных разрядов (в реальном компьютере тоже есть ограничение, например, для типа *Real* в Турбо Паскале — 11-12 цифр), то результат округляется и он равен $0{,}345670*10^4$.

Итак, $a+b=a$ при $b>0$!!! Компьютерная арифметика не совпадает с обычной арифметикой.

На протяжении всей главы используются описанные ниже типы данных, функции и процедуры.

```
Type Real=Extended;
     TPoint=Record
            x, y: Real
            End;
Const_Eps: Real=1e-3;{*Точность вычисления.*}
ZeroPnt: TPoint = (X:0; Y:0);{*Точка
                 с координатами 0,0.*}
```

Реализация операций сравнений: $=$, $<$, $>$, $\le$, $\ge$.

```
Function RealEq(Const a, b: Real): Boolean;
        {*Строго равно.*}
 Begin
  RealEq:=Abs(a-b)<=_Eps;
 End;
```

Остальные функции *RealMore* (строго больше), *RealLess* (строго меньше), *RealMoreEq* (больше или равно), *RealLessEq* (меньше или равно) пишутся по аналогии.

Для определения максимального из двух вещественных чисел приведенные выше функции уже можно использовать.

```
Function RealMax(Const a, b: Real): Real;
     {*Максимальное из двух вещественных чисел.*}
 Begin
  If RealMore(a, b) Then RealMax:=a Else RealMax:=b;
 End;
```

Аналогично пишется функция *RealMin* (поиск минимального числа) и функция *EqPoint* совпадения двух точек на плоскости.

```
Function EqPoint(Const A, B: TPoint):Boolean;
      {*Совпадают ли точки ?*}
 Begin
  EqPoint:=RealEq(A.x, B.x) And RealEq(A.y, B.y);
 End;
```

Функция вычисления расстояния между двумя точками на плоскости (см. рисунок 5.1) имеет вид.

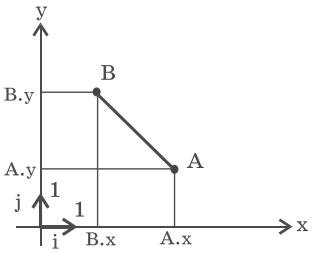


Рис. 5.1

```
Function Dist(Const A, B: TPoint): Real;
          {*Расстояние между двумя точками
           на плоскости.*}
 Begin
  Dist:=Sqrt(Sqr(A.x-B.x) + Sqrt(A.y-B.y));
    {*Вспомним теорему Пифагора – "квадрат
    гипотенузы прямоугольного треугольника равен
    сумме квадратов катетов".*}
 End;
```

Каждую точку плоскости можно считать вектором, начало которого находится в точке (0,0). Обозначим координаты точки (вектора) v через (v_x,v_y), $\boldsymbol{w}=(w_x,w_y)$, $\boldsymbol{q}=(q_x,q_y)$.

Для векторов определены следующие операции:

- сумма: $\boldsymbol{q}=\boldsymbol{v}+\boldsymbol{w}$, $q_x=v_x+w_x$, $q_y=v_y+w_y$;
- разность: $\boldsymbol{q}=\boldsymbol{v}-\boldsymbol{w}$, $q_x=v_x-w_x$, $q_y=v_y-w_y$;
- скалярное произведение: $\boldsymbol{v}\times\boldsymbol{w}=v_xw_x+v_yw_y$;
- векторное произведение: $\boldsymbol{v}\times\boldsymbol{w}=(v_xw_y-v_yw_x)\boldsymbol{k}$.

Скалярным произведением вектора $\boldsymbol{v}$ на вектор $\boldsymbol{w}$ называется число, определяемое равенством

$$\boldsymbol{v}\times\boldsymbol{w}=|v|\times|w|\times cos(\varphi),$$

где символом $|\ |$ обозначается длина вектора, а φ — угол между векторами.

Скалярное произведение обращается в нуль в том и только том случае, когда по крайней мере один из векторов является нулевым или когда векторы $\boldsymbol{v}$ и $\boldsymbol{w}$ ортогональны.

Для скалярного произведения справедливы следующие равенства:

$$(\boldsymbol{v}+\boldsymbol{z})\times\boldsymbol{w}=\boldsymbol{v}\times\boldsymbol{w}+\boldsymbol{v}\times\boldsymbol{z}$$

$$(\lambda\ \boldsymbol{v})\times\boldsymbol{w}\ =\boldsymbol{v}\times(\lambda\ \boldsymbol{w})=\lambda(\boldsymbol{v}\times\boldsymbol{w}).$$

Если векторы заданы своими координатами в ортонормированном базисе $\boldsymbol{i},\boldsymbol{j}$, т. е. $\boldsymbol{v}=(v_x,v_y)$ $\boldsymbol{w}=(w_x,w_y)$, то формула для скалярного произведения с учетом перечисленных свойств записывается

$$\boldsymbol{v}\times\boldsymbol{w}=v_xw_x(\boldsymbol{i}\times\boldsymbol{i})+v_yw_x(\boldsymbol{j}\times\boldsymbol{i})+v_xw_y(\boldsymbol{i}\times\boldsymbol{j})+v_yw_y(\boldsymbol{j}\times\boldsymbol{j})=$$
$$=v_xw_x(\boldsymbol{i}\times\boldsymbol{i})+v_yw_y(\boldsymbol{j}\times\boldsymbol{j})=v_xw_x+v_yw_y.$$

Для скалярного произведения векторов (v,α) и (w,β), изображенных на рисунке 5.2, справедливо соотношение

$$\boldsymbol{v}\times\boldsymbol{w}=v_xw_x+v_yw_y=v\cos(\alpha)\,w\cos(\beta)+v\sin(\alpha)\,w\sin(\beta)=v\,w\cos(\alpha-\beta).$$

При общей начальной точке у двух векторов скалярное произведение больше нуля, если угол между векторами острый, и меньше нуля, если угол тупой.

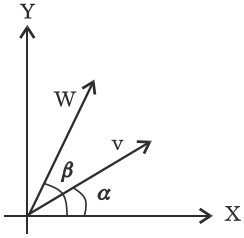


Рис. 5.2

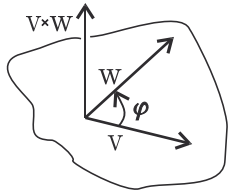


Рис. 5.3

Векторным произведением $\boldsymbol{v}\times\boldsymbol{w}$ называется вектор (см. рисунок 5.3), такой, что:

- длина вектора $v\times w$ равна $|v|\times|w|\times sin(\varphi)$;
- вектор $\boldsymbol{v}\times\boldsymbol{w}$ перпендикулярен векторам $\boldsymbol{v}$ и $\boldsymbol{w}$, т. е. перпендикулярен плоскости этих векторов;
- вектор $\boldsymbol{v}\times\boldsymbol{w}$ направлен так, что из конца этого вектора кратчайший поворот от $\boldsymbol{v}$ к $\boldsymbol{w}$ виден происходящим против часовой стрелки.

Длина векторного произведения численно равна площади параллелограмма, построенного на перемножаемых векторах $\boldsymbol{v}$ и $\boldsymbol{w}$ как на сторонах (см. рисунок 5.4).

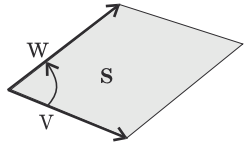


Рис. 5.4

Пусть векторы $\boldsymbol{v}$ и $\boldsymbol{w}$ заданы своими координатами в базисе $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$ (v_x,v_y,v_z) и (w_x,w_y,w_z).

Векторное произведение

$\boldsymbol{v}\times\boldsymbol{w} = (v_x\boldsymbol{i}+v_y\boldsymbol{j}+v_z\boldsymbol{k}, w_x\boldsymbol{i}+w_y\boldsymbol{j}+w_z\boldsymbol{k}) = v_xw_x\,\boldsymbol{i}\times\boldsymbol{i} + v_xw_y\,\boldsymbol{i}\times\boldsymbol{j} + v_xw_z\,\boldsymbol{i}\times\boldsymbol{k} + v_yw_x\,\boldsymbol{j}\times\boldsymbol{i} + v_yw_y\,\boldsymbol{j}\times\boldsymbol{j} + v_yw_z\,\boldsymbol{j}\times\boldsymbol{k} + v_zw_x\,\boldsymbol{k}\times\boldsymbol{i} + v_zw_y\,\boldsymbol{k}\times\boldsymbol{j} + v_zw_z\,\boldsymbol{k}\times\boldsymbol{k}.$

Векторные произведения координатных ортов равны:

$\boldsymbol{i}\times\boldsymbol{i}=0,\ \boldsymbol{j}\times\boldsymbol{j}=0,\ \boldsymbol{k}\times\boldsymbol{k}=0,\ \boldsymbol{i}\times\boldsymbol{j}=\boldsymbol{k},\ \boldsymbol{j}\times\boldsymbol{i}=-\boldsymbol{k},\ \boldsymbol{j}\times\boldsymbol{k}=\boldsymbol{i},\ \boldsymbol{k}\times\boldsymbol{j}=-\boldsymbol{i},\ \boldsymbol{k}\times\boldsymbol{i}=\boldsymbol{j}$ и $\boldsymbol{i}\times\boldsymbol{k}=-\boldsymbol{j}$.

Формула имеет вид:

$$\boldsymbol{v}\times\boldsymbol{w} = (v_yw_z-v_zw_y)\boldsymbol{i} + (v_zw_x-v_xw_z)\boldsymbol{j} + (v_xw_y-v_yw_x)\boldsymbol{k}.$$

Ее можно записать в другом, более компактном виде (через определитель):

$$\boldsymbol{v}\times\boldsymbol{w} = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix}$$

Раскрывая определитель, мы получаем ту же самую формулу. Учитывая наш случай, а именно то, что координаты v_z и w_z векторов равны нулю, получаем $\boldsymbol{v}\times\boldsymbol{w}=(v_xw_y-v_yw_x)\boldsymbol{k}$. После несложных преобразований получаем

$$\boldsymbol{v}\times\boldsymbol{w} = (v\times\cos(\alpha)\times w\times\sin(\beta)-v\times\sin(\alpha)\times w\times\cos(\beta))\boldsymbol{k} = (v\times w\times\sin(\beta-\alpha))\boldsymbol{k}.$$

Векторное произведение ненулевых векторов равно нулю тогда и только тогда, когда векторы параллельны.

Пример. Найти площадь треугольника *OAB* (см. рисунок 5.5).

Очевидно, что его площадь равна половине площади параллелограмма *OACB*, а площадь последнего равна модулю векторного произведения векторов *OA* и *OB*. Находим, $S_{OACB}=A_x\times B_y-A_y\times B_x$, отсюда

$$S_{OAB}=\frac{A_x\times B_y - A_y\times B_x}{2}.$$

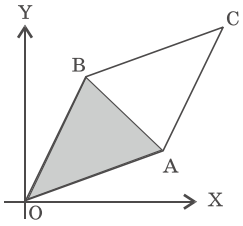


Рис. 5.5

Отметим, что значение S_{OBA} имеет другой знак (ориентированная площадь).

Тип данных для описания векторов имеет вид:

```
Type TVecCart=TPoint; (*Record x, y: Real End*)
```

Рассмотрим процедуры работы с векторами в декартовой системе координат.

```
Procedure AddVecCart(Const a, b: TVecCart;
```

```
  Var c:TVecCart);
{*Сумма двух векторов в декартовой системе координат.*}
   Begin
    c.x:=a.x+b.x;c.y:=a.y+b.y;
   End;
```

Аналогично пишутся и процедуры вычисления разности двух векторов *SubVecCart*, умножения вектора на число *MulKVecCart*, а также функции нахождения скалярного произведения *SkMulCart* и векторного произведения *VectMulCart* векторов в декартовой системе координат. Например:

```
  Function SkMulCart (Const a, b: TVecCart): Real;
           {* Скалярное произведение.*}
   Begin
    SkalarMulCart:=a.x*b.x+a.y*b.y;
   End;
```

Вектор v можно задать в полярной системе координат через его длину (модуль) v и угол α относительно оси X. Координаты полярной системы координат (v, α) и прямоугольной декартовой (v_x,v_y) связаны соотношениями:

$v_x=v\times cos(\alpha)$, $v_y=v\times sin(\alpha)$,

$v=Sqrt(v_x^2+v_y^2)$, $tan(\alpha)=v_y/v_x$.

Тип данных для описания вектора в полярной системе координат имеет вид

```
  Type TVecPol=Record
               rst, angle: Real
               End;
```

Операции перевода из одной системы координат в другую реализуются с помощью следующих процедур и функций. Функция определения угла используется при переводе в полярную систему координат.

```
  Function GetAngle(Const x, y: Real): Real;
    {*Возвращает угол от 0 до 2*Pi в радианах.*}
   Var rs, c: Real;
   Begin
    rs:=Sqrt(Sqr(x) + Sqr(y));{*Вычисляется
           расстояние от точки (0,0) до (x,y). Можно
           воспользоваться функцией  Dist.*}
```

```
    If RealEq(rs, 0)
      Then GetAngle:=0
      Else
        Begin
          c:=x/rs;
          If RealEq(c, 0)
            Then c:=Pi/2
            Else
              c:=ArcTan(Sqrt(Abs(1-Sqr(c))) /c);
          If RealLess(c, 0) Then c:=Pi+c;
          If RealLess(y, 0) Then c:=2*Pi-c;
          GetAngle:=c;
        End;
  End;
Procedure CartToPol(Const a: TVecCart;
  Var b:TVecPol);{*Перевод из декартовой системы
              координат в полярную.*}
  Begin
    b.rst:=Sqrt(Sqr(a.x) + Sqr(a.y));
    b.angle:=GetAngle(a.x, a.y);
  End;
Procedure PolToCart(Const a: TVecPol;
  Var b: TVecCart);
            {*Перевод из полярной системы координат
             в декартовую.*}
  Begin
    b.x:=a.rst*cos(a.angle);
    b.y:=a.rst*sin(a.angle);
  End;
```

5.2. Прямая линия и отрезок прямой

Прямая линия на плоскости (см. рисунок 5.6), проходящая через две точки (v_x,v_y) и (w_x,w_y), определяется следующим линейным уравнением от двух переменных:

$$(w_x-v_x)\times(y-v_y)=(w_y-v_y)\times(x-v_x).$$

После преобразований получаем:

$$-(w_y-v_y)\times x+(w_x-v_x)\times y+(w_y-v_y)\times v_x-(w_x-v_x)\times v_y=0$$

или, после соответствующих обозначений:

$$A\times x+B\times y+C=0,$$

где $A=v.y-w.y$, $B=w.x-v.x$, $C=-(v.x\times(v.y-w.y)+v.y\times(w.x-v.x))$.

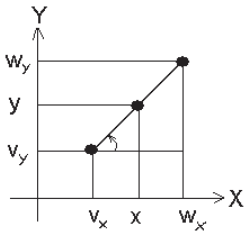


Рис. 5.6

Параметры прямой линии описываются с помощью следующего типа данных:

```
Type TLine=Record
           A, B, C: Real
           End;

Procedure Point2ToLine(Const v, w: TPoint;
  Var L:TLine);
  {*Определение уравнения прямой
           по координатам двух точек.*}
  Begin
   L.A:= v.y - w.y;
   L.B:= w.x -v.x;
   L.C:=-(v.x*L.A + v.y*L.B);
  End;
```

Если прямые заданы с помощью уравнений $A_1 \times x + B_1 \times y + C_1 = 0$ и $A_2 \times x + B_2 \times y + C_2 = 0$, то точка их пересечения, в случае ее существования, определяется по формулам ($A_1 \times B_2 - A_2 \times B_1 <> 0$):

$$x = (C_1 \times B_2 - C_2 \times B_1)/(A_1 \times B_2 - A_2 \times B_1),$$
$$y = (A_1 \times C_2 - A_2 \times C_1)/(A_1 \times B_2 - A_2 \times B_1)$$

обычным решением системы двух уравнений с двумя неизвестными.

Функция *CrossLine* определяет существование точки пересечения (с учетом тех погрешностей, которые определяются введенной точностью вычислений), а в процедуре *Line2ToPoint* вычисляются координаты точки пересечения.

```
Function CrossLine(Const L1, L2: TLine): Boolean;
      {*Определяет существование точки пересечения
       двух прямых. Значение функции равно True,
       если точка есть, и False, если прямые
       параллельны.*}
```

```
 Var st: Real;
 Begin
  st:=L1.A*L2.B-L2.A*L1.B;
  CrossLine:=Not RealEq(st,0);
 End;
Procedure Line2ToPoint(Const L1, L2: Tline;
 Var P:TPoint);{*Определение координат точки
                пересечения двух линий.*}
 Var st: Real;
 Begin
  st:=L1.A*L2.B-L2.A*L1.B;
  P.X:=(L1.C*L2.B-L2.C*L1.B)/st;
  P.Y:=(L1.A*L2.C-L2.A*L1.C)/st;
 End;
```

Координаты точек отрезка можно задать двумя параметрическими уравнениями от одной независимой переменной t:

$$x=v_x+(w_x-v_x)\times t,$$
$$y=y_x+(w_y-v_y)\times t.$$

При $0\le t\le 1$ точка (x,y) лежит на отрезке, а при $t<0$ или $t>1$ — вне отрезка на прямой линии, продолжающей отрезок.

Для проверки принадлежности точки P отрезку необходимо выполнение равенства в уравнении $(w_x-v_x)\times(y-v_y)=(w_y-v_y)\times(x-v_x)$ и принадлежность координаты, например, x точки P интервалу, определяемому координатами x концов отрезков.

```
Function AtSegm(Const A, B, P: TPoint): Boolean;
     {*Проверка принадлежности точки P отрезку AB.*}
 Begin
  If EqPoint(A, B)
    Then AtSegm:=EqPoint(A, P)
          {*Точки A и B совпадают, результат
           определяется совпадением точек A и P.*}
    Else
     AtSegm:=RealEq((B.x-A.x)*(P.y-A.y), (B.y-A.y)*
         (P.x-A.x)) And ((RealMoreEq(P.x, A.x) And
          RealMoreEq(B.x,P.x))
          Or (RealMoreEq(P.x, B.x)
          And RealMoreEq(A.x, P.x)));
 End;
```

Задание. Измените функцию для случая, когда отрезок описывается параметрическими уравнениями.

Процедура, с помощью которой определяются координаты точки пересечения двух отрезков, пишется на основе ранее приведенных процедур построения прямой по двум точкам и нахождения точки пересечения прямых (не утверждается то, что точка пересечения принадлежит отрезкам).

```
Procedure FindPointCross(Const fL, fR, sL, sR:
   TPoint; Var rs: TPoint);{*Если точки пересечения
       не существует, то значение rs равно (0,0).*}
 Var L1, L2: TLine;
 Begin
  Point2ToLine(fL, fR, L1);
  Point2ToLine(sL, sR, L2);
  If CrossLine(L1,L2)
    Then Line2ToPoint(L1, L2, rs)
    Else rs:=ZeroPnt;
 End;
```

Примечание

Процедуру следует доработать в том случае, если точка пересечения прямых совпадает с точкой (0,0).

Пусть два отрезка находятся на одной прямой. Требуется определить их взаимное расположение.

```
Function SegmLineCross(Const fL, fR, sL, sR:
Tpoint): Byte; {*Отрезки находятся на одной
       прямой, проверка их взаимного расположения.
       Результат равен 0, если отрезки пересекаются
       в одной точке, 1 — не имеют пересечения и 2 —
       есть пересечение более чем в одной точке.*}
 Var Minf, Maxf,Mins,Maxs: Real;
 Begin
  Minf:=RealMin(Dist(fL,ZeroPnt),Dist(fR,ZeroPnt));
  Maxf:=RealMax(Dist(fL,ZeroPnt),Dist(fR,ZeroPnt));
  Mins:=RealMin(Dist(sL,ZeroPnt),Dist(sR,ZeroPnt));
  Maxs:=RealMax(Dist(sL,ZeroPnt),Dist(sR,ZeroPnt));
  If RealEq(Minf,Maxs) Or RealEq(Maxf,Mins)
    Then SegmLineCross:=0
    Else
If RealMore(Mins,Maxf) Or RealMore(Minf,Maxs)
       Then SegmLineCross:=1
       Else SegmLineCross:=2;
 End;
```

Даны два отрезка на плоскости, заданные координатами своих концов. Определить их взаимное расположение (до написания функции попробуйте прорисовать все возможные случаи).

```
Function SegmCross(Const fL, fR, sL, sR: Tpoint): Byte; {* Результат равен 0, если отрезки пересекаются в одной точке и лежат на одной прямой, 1 – не имеют пересечения и лежат на одной прямой, 2 – отрезки лежат на одной прямой и есть пересечение более чем в одной точке, 3 – отрезки лежат на параллельных прямых, 4 – отрезки не лежат на одной прямой и не имеют точки пересечения, 5 – отрезки не лежат на одной прямой и пересекаются на концах отрезков, 6 – отрезки не лежат на одной прямой, точка пересечения принадлежит одному из отрезков и является концом другого отрезка, 7 – отрезки не лежат на одной прямой и пересекаются в одной точке, не совпадающей ни с одним концом отрезков.*}
  Var rs:TPoint;
     L1,L2:TLine;
  Begin
   Point2ToLine(fL, fR, L1);
   Point2ToLine(sL, sR, L2);
   If CrossLine(L1,L2)
    Then
     Begin
       Line2ToPoint(L1, L2, rs)
       If EqPoint(rs,fL) Or EqPoint(rs,fR) Or
          EqPoint(rs,sL) Or EqPoint(rs,sR)
        Then SegmCross:=5
        Else
         If AtSegm(fL,fR,rs) And AtSegm(sL,sR,rs)
          Then SegmCross:= 7
          Else
           If AtSegm(fL,fR,rs) Or AtSegm(sL,sR,rs)
            Then SegmCross:=6
            Else SegmCross:=4;
     End
    Else
      If EqPoint(L1.A*L2.B,L2.A*L1.B) And Not
        (EqPoint(L1.C,L2.C))
       Then SegmCross:=3
       Else SegmCross:= SegmLineCross( fL, fR,
              sL, sR);
  End;
```

Задачу о пересечении отрезков на плоскости можно (естественно) решать по-другому. На рисунке 5.7 показаны два отрезка. Найдем ряд векторных произведений, они показаны на рисунке 5.8.

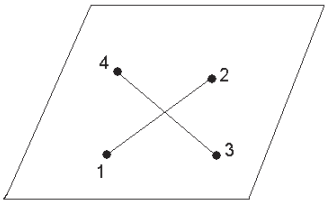
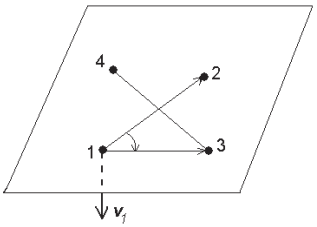
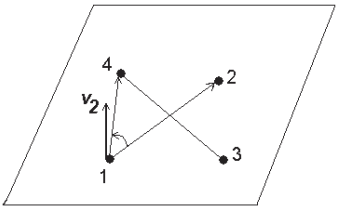


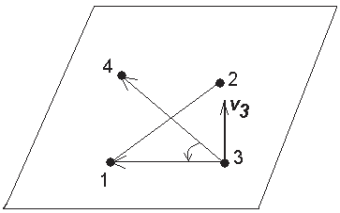
Рис. 5.7



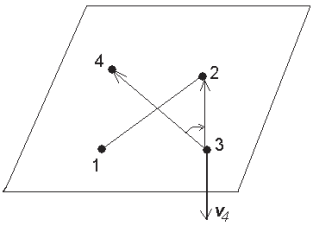
(а)



(б)



(в)



(г)

Рис. 5.8

Рассмотрим произведения длин v_1, v_2, v_3, v_4. Очевидно, что если $v_1 \times v_2 < 0$ и $v_3 \times v_4 < 0$, то отрезки пересекаются (в произведениях берутся длины векторов v_1, v_2, v_3, v_4). При $v_1 \times v_2 > 0$ или $v_3 \times v_4 > 0$ отрезки не пересекаются.

Рассмотрим варианты, при которых одна из длин равна нулю. Если $v_1 = 0$, $v_2 <> 0$ и $v_3 \times v_4 < 0$, то третья точка лежит на первом отрезке. При $v_2 = 0$ и аналогичных условиях четвертая точка лежит на первом отрезке, $v_3 = 0$ — первая на втором отрезке и при $v_4 = 0$ — вторая точка на втором отрезке. Особый случай, когда $v_1 = 0$, $v_2 = 0$, $v_3 = 0$, $v_4 = 0$. Отрезки расположены на

одной прямой, необходимы дополнительные проверки для выяснения того, перекрываются они или нет. Все ли особые случаи рассмотрены?

Задание. Определите значения векторных произведений при совпадении каких-либо двух точек, например 2-й и 3-й.

Задание. Разработайте процедуру определения взаимного расположения двух отрезков на плоскости с использованием рассмотренных векторных произведений.

Еще один вариант решения задачи о пересечении отрезков возможен при использовании параметрической формы записи уравнения прямой. Отрезки заданы координатами своих концов:

(x_0,y_0) – (x_1,y_1) и (x_2,y_2) – (x_3,y_3).

Уравнения имеют вид:

$x=x_0+t_1\times(x_1-x_0)$, $y=y_0+t_1\times(y_1-y_0)$,
где $t_1\in[0,1]$, и $x=x_2+t_2\times(x_3-x_2)$, $y=y_2+t_2\times(y_3-y_2)$, где $t_2\in[0,1]$.

В точке пересечения, при ее существовании выполняются равенства:

$$x_0+t_1\times(x_1-x_0) = x_2+t_2\times(x_3-x_2)$$
$$y_0+t_1\times(y_1-y_0) = y_2+t_2\times(y_3-y_2).$$

Есть система двух уравнений с двумя неизвестными t_1 и t_2. Если существует единственное решение и t_1, $t_2\in(0,1)$, то отрезки пересекаются. Если одно из значений t_1 и t_2 равно 0 или 1, а второе принадлежит [0,1], то отрезки пересекаются в одной из вершин, в остальных случаях отрезки не пересекаются.

Задание. Разработайте процедуру, определяющую пересекаются ли отрезки. На основании выше изложенного материала.

Нормальным вектором n к прямой L ($A\times x+B\times y+C=0$) называют всякий ненулевой вектор, перпендикулярный этой прямой (см. рисунок 5.9). Известно, что координаты вектора определяются значениями A и B в уравнении линии $L-\boldsymbol{n}=(A,B)$. Для того, чтобы найти уравнение прямой на плоскости L (см. рисунок 5.10, проходящей через заданную точку M_0 перпендикулярно заданному направлению $\boldsymbol{n}=(A,B)$, необходимо раскрыть скалярное произведение $(\boldsymbol{r}-\boldsymbol{r}_0, \boldsymbol{n})=0$. Оно равно $A\times(x-x_0)+B\times(y-y_0)=0$.

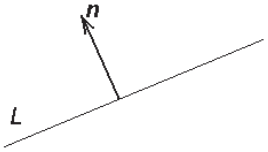


Рис. 5.9

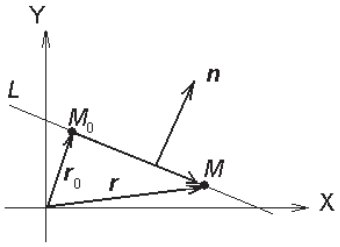


Рис. 5.10

```
Procedure PerLine(Const n: TLine; Const P: Tpoint;
Var L: TLine);
{*Линия, перпендикулярная n,
проходящая через точку P.*}
 Begin
   L.A:=n.B;
   L.B:=-n.A;
   L.C:=-L.A*P.X-L.B*P.Y
 End;
```

Расстояние от точки $P(x_0,y_0)$ до прямой L вычисляется по формуле: $\rho(P,L) = \dfrac{|L.A \times x_0 + L.B \times y_0 + L.C|}{\sqrt{L.A^2 + L.B^2}}$

```
Function DistPointToLine(Const P: TPoint;
Const L:TLine): Real;{*Расстояние от точки до
прямой.*}
 Begin
  DistPointToLine:=Abs((L.A*P.x+L.B*P.y+L.C))
       /Sqrt(Sqr(L.A)+Sqr(L.B));
 End;
```

5.3. Треугольник

Даны три отрезка *a*, *b*, *c*. Для существования треугольника с такими длинами сторон требуется одновременное выполнение условий **(*a*+*b*>*c*)**, **(*b*+*c*>*a*)** и **(*a*+*c*>*b*)**.

```
Function IsTrian(Const a, b, c: Real): Boolean;
         {*Проверка существования треугольника
          со сторонами a, b, c.*}
 Begin
  IsTrian:=RealMore(a+b, c) And RealMore(a+c, b)
      And RealMore(b+c, a);
 End;
```

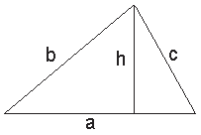


Рис. 5.11

Площадь треугольника (см. рисунок 5.11). Если известны длины сторон треугольника, то для вычисления площади треугольника обычно используется формула Герона

$$S=\sqrt{p\times(p-a)\times(p-b)\times(p-c)}$$

где $p=(a+b+c)/2$.

```
Function Sq(a,b,c:Real):Real;
 Var p:Real;
 Begin
  p:=(a+b+c)/2;
  Sq:=Sqrt(p*(p-a)*(p-b)*(p-c));
 End;
```

Возможно использование и других формул:

$$S=\frac{a\times h}{2}=\frac{a\times b\times\sin c}{2}=\frac{a^2\times\sin B\times\sin C}{2\times\sin A}=\frac{h^2\times\sin A}{2\times\sin B\times\sin C},$$

где большими буквами A, B, C обозначены углы против соответствующих сторон.

Гораздо интереснее второй способ вычисления площади треугольника — через векторное произведение. Длина вектора дает удвоенную площадь треугольника, построенного на этих векторах.

Пусть даны три точки $p_1=(x_1,y_1)$, $p_2=(x_2,y_2)$, $p_3=(x_3,y_3)$ на плоскости, определяющие вершины треугольника. Совместим начало координат с первой точкой.

Векторное произведение равно $\begin{pmatrix} i & j & k \\ x_2-x, & y_2-y, & 0 \\ x_3-x, & y_3-y, & 0 \end{pmatrix}$.

Вычисление длины вектора равносильно раскрытию определителя $\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1\times(y_2-y_3)-y_1\times(x_2-x_3)+(x_2\times y_3-x_3\times y_2)$ (раскрываем по первой строке).

```
Function SquareTrian(Const p1, p2, p3: TPoint):
  Real; {*Вычисление ориентированной площади
        треугольника.*}
 Begin
  SquareTrian:=(p1.x*(p2.y-p3.y)-p1.y(p2.x-p3.x)
    + (p2.x*p3.y-p3.x*p2.y))/2;
 End;
```

Пример. Вычислим удвоенную площадь треугольника, изображенного на рисунке 5.12.

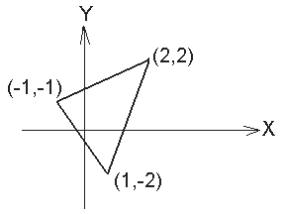


Рис. 5.12

Значение определителя $\begin{vmatrix} 2 & 2 & 1 \\ -1 & -1 & 1 \\ 1 & -2 & 1 \end{vmatrix}$ равно 9.

Значение определителя $\begin{vmatrix} 2 & 2 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & 1 \end{vmatrix}$ равно −9. Отличие: в первом случае угол считался против часовой стрелки, во втором — по часовой стрелке.

Пример. Рисунок 5.13.

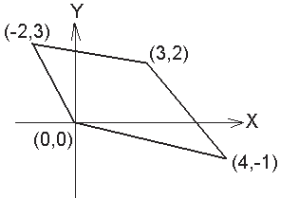


Рис. 5.13

Определитель $\begin{vmatrix} 4 & -1 & 1 \\ 0 & 0 & 1 \\ -2 & 3 & 1 \end{vmatrix}$ равен 10,

а определитель $\begin{vmatrix} 4 & -1 & 1 \\ 3 & 2 & 1 \\ -2 & 3 & 1 \end{vmatrix}$ равен 15.

Вывод: при обходе против часовой стрелки получаем положительные величины, а при обходе по часовой стрелке — отрицательные значения.

Замечательные линии и точки в треугольнике.

1. *Высоту* треугольника, опущенную на сторону a, обозначим через h_a (см. рисунок 5.14). Через три стороны она выражается формулой

$$h_a = \frac{2 - \sqrt{p \times (p-a) \times (p-b) \times (p-c)}}{a},$$

где $p=(a+b+c)/2$.

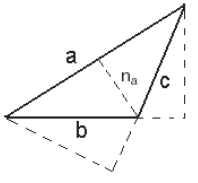


Рис. 5.14

```
Function GetHeight(Const a, b, c: Real):
Real;{*Вычисление длины высоты, проведенной
из вершины, противоположной стороне треугольника
с длиной a.*}
 Var p: Real;
 Begin
  p:=(a+b+c)/2;
```

```
  GetHeight:=2*Sqrt(p*(p-a)*(p-b)*(p-c))/a;
End;
```

2. *Медианы* треугольника пересекаются в одной точке (всегда внутри треугольника), являющейся центром тяжести треугольника. Эта точка делит каждую медиану в отношении 2:1, считая от вершины. Медиану на сторону *a* (см. рисунок 5.15) обозначим через m_a. Через три стороны треугольника она выражается формулой

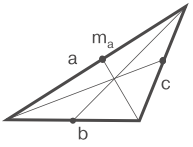


Рис. 5.15

$$m_a = \frac{\sqrt{2 \times b^2 + 2 \times c^2 - a^2}}{2}.$$

```
Function GetMed(Const a, b, c:
Real): Real; {*Вычисление длины
медианы, проведенной из вершины,
противоположной стороне треугольника с длиной a.*}
 Begin
  GetMed:=Sqrt(2*(b*b+c*c)-a*a)/2;
 End;
```

3. Три *биссектрисы* треугольника пересекаются в одной точке (всегда внутри треугольника), являющейся центром вписанного круга (см. рисунок 5.16). Его радиус вычисляется по формуле

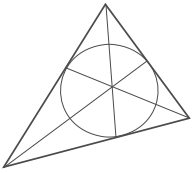


Рис. 5.16

$$r = \sqrt{\frac{(p-a) \times (p-b) \times (p-c)}{p}}.$$

Биссектрису к стороне *a* обозначим через β_a, ее длина выражается формулой

$$\beta_a = \frac{2 \times \sqrt{b \times c \times p \times (p-a)}}{b+c},$$ где p — полупериметр.

```
Function GetRadIns(Const a, b, c: Real): Real;
{*Вычисление радиуса окружности, вписанной
в треугольник с длинами сторон a, b, c.*}
 Var p: Real;
 Begin
  p:=(a+b+c)/2;
  GetRadIns:=Sqrt((p-a)*(p-b)*(p-c)/p);
 End;
```

```
Function GetBis(Const a, b, c: Real): Real;
{*Вычисление длины биссектрисы, проведенной
из вершины, противоположной стороне треугольника
с длиной a.*}
 Var p: Real;
 Begin
  p:=(a+b+c)/2;
  GetBis:=2*Sqrt(b*c*p*(p-a))/(b+c);
 End;
```

4. Три *перпендикуляра к сторонам* треугольника, проведенные через их середины, пересекаются в одной точке, служащей центром описанного круга (см. рисунок 5.17). В тупоугольном треугольнике эта точка лежит вне треугольника, в остроугольном — внутри, в прямоугольном — на середине гипотенузы. Его радиус вычисляется по формуле

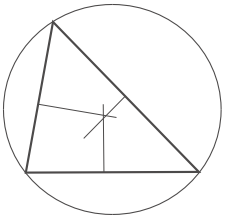


Рис. 5.17

$$R = \frac{a \times b \times c}{4\sqrt{p \times (p-a) \times (p-b) \times (p-c)}}$$

В равнобедренном треугольнике высота, медиана и биссектриса, опущенные на основание, а также перпендикуляр, проведенный через середину основания, совпадают друг с другом; в равностороннем то же имеет место для всех трех сторон. В остальных случаях ни одна из упомянутых линий не совпадает с другой. Центр тяжести, центр вписанного круга и центр описанного круга совпадают друг с другом только в равностороннем треугольнике.

```
Function GetRadExt(Const a, b, c: Real): Real;
{*Вычисление радиуса окружности, описанной около
треугольника с длинами сторон a, b, c.*}
 Var p: Real;
 Begin
  p:=(a+b+c)/2;
  GetRadExt:=a*b*c/(4*Sqrt(p*(p-a)*(p-b)*(p-c)));
 End;
```

5.4. Многоугольник

Многоугольник назовем простым, если никакая пара непоследовательных его ребер не имеет общих точек. Требуется определить, является ли заданный многоугольник простым.

Для решения этой задачи у нас разработан аппарат, ибо суть решения в определении взаимного расположения сторон многоугольника. Если нет ни одного пересечения сторон, включая случай их частичного наложения, то многоугольник простой.

Запрещенные ситуации пересечения сторон многоугольника показаны на рисунке 5.18 (цифрами обозначены концы отрезков, соответственно первого и второго).

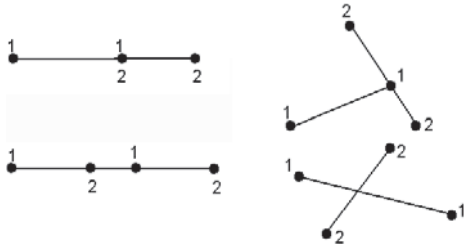


Рис. 5.18

```
Function IsPoligonSimple
(Const A:Array Of TPoint;Const N:Word):Boolean;
{*Проверка простоты многоугольника. Значение функции
равно False, если многоугольник не является простым, и
True, если он простой. Точки в массиве A заданы в поряд-
ке обхода по часовой или против часовой стрелки. Пред-
полагаем, что если отрезки лежат на одной прямой и пе-
ресекаются в одной точке, то это один отрезок.*}
  Var i, j: Integer;
      pp:Boolean;
  Begin
   pp:=True;
   i:=1;
   While (i<=N-1) And pp Do
    Begin
      j:=i+1;
      While (j<=N) And pp Do
        Begin
         Case SegmCross(A[i],A[i+1],A[j],A[(j+1) Mod N])
          Of{*Функция проверки взаимного
          расположения двух отрезков описана ранее.*}
          0,2,6,7: pp:=False;
        End;
      Inc(j);
   End;
```

```
    Inc(i);
  End;
   IsPoligonSimple:=pp;
 End;
```

Задача. Дана точка p и простой N-угольник Q. Определить, принадлежит ли точка p внутренней области простого N-угольника Q, включая его стороны.

Проведем через точку p прямую L, параллельную оси X. Если L не пересекает Q, то p не принадлежит Q, она внешняя по отношению к Q. Найдем количество пересечений (w) луча, выходящего из точки p влево или вправо. Точка p лежит внутри Q тогда и только тогда, когда w нечетно. Это общий (идеальный) случай. В этой, как и в любой геометрической задаче, есть частные случаи.

На рисунке 5.19 для первой точки все хорошо, она внешняя, количество пересечений четно (считаем вправо). Для второй и третьей точек ситуация чуть хуже — на прямой L лежит одна из сторон многоугольника; прямая L проходит через вершину многоугольника.

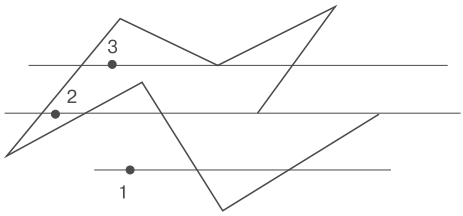


Рис. 5.19

Вырожденные случаи. Лучше повернуть чуть-чуть прямую L против часовой стрелки, что устранит вырожденность и позволит считать по схеме, приведенной ниже.

```
Function IncludPoint(Const A: Array Of TPoint;
Const N: Word; Const P: TPoint): Boolean;
{*Проверка принадлежности точки многоугольнику
(включая стороны).*}
 Var i, nxt, sc: Integer;
     lf, rg: TPoint;
     nx: Real;
 Begin
  sc:=0; IncludPoint:=True;
  For i:=0 To N-1 Do
```

```
Begin {*Для каждой стороны
        многоугольника находим координату x точки
        пересечения с горизонтальным лучом.*}
  nxt:=(i+1) Mod N;
  If AtSegm(A[i], A[nxt], P) Then Exit;{*Проверка
       принадлежности точки P отрезку
      (A[i],A[nxt]).*}
  lf:=A[i]; rg:=A[i];
  If RealLess(A[i].y, A[nxt].y)
    Then rg:=A[nxt]
    Else lf:=A[nxt];
  If RealLess(lf.y, A[i].y) And RealLessEq
     (A[i].y, rg.y) Then
   Begin
    Nx:=((rg.x-lf.x)*P.y+lf.x*rg.y-lf.y*rg.x)/
        (rg.y-lf.y); {*Вычисление координаты x
        точки пересечения стороны многоугольника
        с прямой, параллельной оси X. Для решения
        задачи достаточно в уравнение
        (wx-vx)*(y-vy)=(wy-vy)*(x-vx) подставить
        координаты соответствующих точек, раскрыть
        скобки и сократить подобные члены.*}
    If RealMore(nx, P.x) Then Inc(sc);
   End;
 End;
 IncludPoint:=Odd(sc);
End;
```

Задача. Вычислить площадь простого плоского многоугольника.

Метод. Соединяем начало координат отрезками с вершинами многоугольника. Последовательно вычисляем ориентированные площади треугольников. В результате получим площадь нашего многоугольника.

Рассмотрим задачу на примере. Удвоенная площадь прямоугольного треугольника вычисляется по формуле $a \times b$ (a,b — стороны треугольника, образующие прямой угол), а трапеции — $(a+b) \times h$, где a, b — длины оснований, h — высота. Пусть наш многоугольник является четырехугольником $p_0p_1p_2p_3$, где $p_i=(x_i,y_i)$. Вычислим площадь треугольников, показанных на рисунке 5.20:

$$S_1=x_1 \times y_1-x_0 \times y_0-(y_1+y_0) \times (x_1-x_0)=x_0 \times y_1-x_1 \times y_0$$
$$S_2=x_2 \times y_2-x_1 \times y_1-(y_2+y_1) \times (x_2-x_1)=x_1 \times y_2-x_2 \times y_1$$

$S_3=x_3\times y_3-x_2\times y_2-(y_3+y_2)\times(x_3-x_2)=x_2\times y_3-x_3\times y_2$
$S_4=x_0\times y_0-x_3\times y_3-(y_0+y_3)\times(x_0-x_3)=x_3\times y_0-x_0\times y_3$

Мы могли бы это же самое сделать с помощью векторных произведений, рассмотренных выше (эта формула уже получена ранее), но пусть будет так — другой способ.

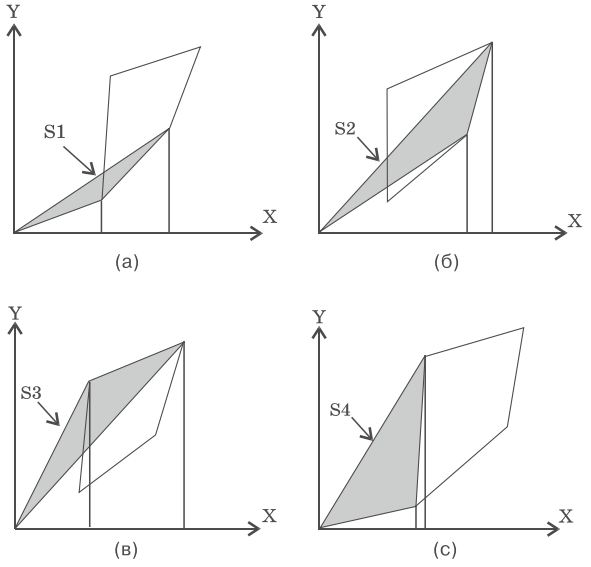


Рис. 5.20

После суммирования и группировки получаем:

$S_1+S_2+S_3+S_4=x_1\times(y_2-y_0)+x_2\times(y_3-y_1)+x_3\times(y_0-y_2)+x_0\times(y_1-y_3)$.

Убедитесь в правильности наших рассуждений на следующем примере, изображенном на рисунке 5.21. Площадь многоугольника (S) равна 3,5. По нашим формулам

$2\times S=1\times(1-2)+4\times(4-1)+1\times(3-1)+1\times(3-4)+$
$+2\times(2-3)+2\times(2-3)+1\times(1-2)=7.$

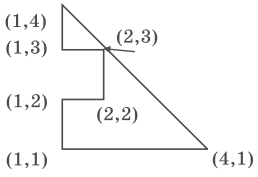


Рис. 5.21

После этих рассуждений и примеров можно написать функцию вычисления площади простого многоугольника.

```
Function Square(Const A: Array Of TPoint;
Const N:Word): Real;{*Ориентированная площадь много-
угольника из N точек.*}
   Var  i: Word;
```

```
      sc: Real;
  Begin
   If N<3
     Then Square:=0
     Else
       Begin
         Sc:=A[0].x*(A[1].y-A[N-1].y)+A[N-1].x*
            (A[0].y-A[N-2].y);
         For i:=1 To N-2 Do
         sc:=sc+A[i].x*(A[i+1].y-A[i-1].y);
         Square:=sc/2;
       End;
  End;
```

Задача. Дан многоугольник. Определить, является ли он выпуклым.

Многоугольник выпуклый, если все диагонали лежат внутри него. Сумма внутренних углов в выпуклом многоугольнике равна $180°\times(N-2)$, где N — число сторон многоугольника.

Решить задачу определения выпуклости можно и по-другому. Все треугольники, образованные тройками соседних вершин в порядке их обхода, имеют одну ориентацию. При этом следует учесть тот факт, что нахождение тройки вершин на одной прямой не нарушает факта выпуклости. Первые два многоугольника на рисунке 5.22 выпуклые, третий, независимо от направления обхода, не является выпуклым.

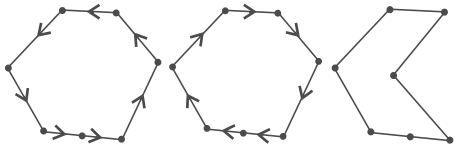


Рис. 5.22

```
Function IsPoligonConvex(Const A: Array Of TPoint;
Const N: Word): Boolean;
{*Проверка выпуклости многоугольника, значение функ-
ции равно True, если многоугольник выпуклый.*}
   Var  bn, nw: Byte;
        i: Integer;rp: Real;
   Begin
    IsPoligonConvex:=False;
    If N>3 Then
     Begin
```

```
    bn:=1;
    For i:=0 To N-1 Do
      Begin
        rp:=SquareTrian(A[(i+N-1) Mod N], A[i], A[(i+1)
          Mod N]); {*Ориентированная площадь
             треугольника, построенного по трем
             соседним вершинам многоугольника.*}
        If RealEq(rp, 0)
          Then nw:=1{*Точки находятся на одной прямой.*}
          Else
            If RealLess(rp, 0) Then nw:=0 Else nw:=2;
        If (bn=1) Then bn:=nw
                  Else
          If (nw<>1) And (nw<>bn) Then Exit;{*Вершины
            многоугольника лежат не на одной прямой,
            ориентация треугольников не совпадает -
            многоугольник не выпуклый.*}
      End;
  End;
  IsPoligonConvex:=True;
End;
```

5.5. Выпуклая оболочка*

На плоскости задано множество S, содержащее N точек. Требуется построить их выпуклую оболочку.

Идея алгоритма Грэхема. Тройки последовательных точек проверяются в порядке обхода против часовой стрелки на предмет того, образуют ли они угол, больший или равный π. Если угол $q_1q_2q_3$ больше или равен π, то считают, что $q_1q_2q_3$ образуют «правый поворот», иначе они образуют «левый поворот».

Алгоритм (см. рисунок 5.23):

- найти внутреннюю точку t;
- используя t как начало координат, отсортировать точки множества S по неубыванию в соответствии с полярным углом и расстоянием от t;
- выполнить обход точек, исключая точки типа q_2 образующие «правый поворот».

Этот алгоритм можно упростить. Можно находить не внутреннюю точку, а самую левую и верхнюю. Она заведомо при-

---

* Идеи всех алгоритмов этого параграфа взяты из прекрасной книги Ф. Препарата и М Шеймоса [24].

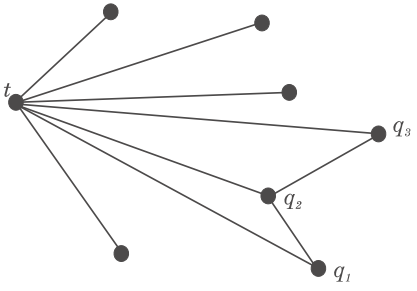


Рис. 5.23

надлежит выпуклой оболочке. При этом значения углов вычисляются относительно этой точки.

Известно, что сортировка N элементов может быть выполнена за время $O(N \times logN)$ (методы Хоара, слияния, пирамидальной сортировки). Это самая трудоемкая по времени часть алгоритма. Обход может быть выполнен за время, пропорциональное N. Значит, выпуклую оболочку можно построить за время, пропорциональное $N \times logN$.

Опишем структуры данных.

```
Var A:Array[1..MaxN] Of TPoint;{*Координаты точек.*}
  N: Integer;{*Количество точек.*}
  M: Integer;{*Количество точек в выпуклой
             оболочке.*}
  ls: Array[1..MaxN] Of Integer;{*Номера точек.
           В процедуре типа Init необходимо
           выполнить следующее присваивание:
           For i:=1 To N Do ls[i]:=i.*}
  bb: Array[1..MaxN] Of Boolean;{*Признак
           принадлежности точки выпуклой
           оболочке, начальное присвоение -
           FillChar(bb, SizeOf(bb), True), все
           точки принадлежат выпуклой оболочке.*}
  Rd: Array[1..MaxN] Of Real;{*Оценки углов.*}
```

Фрагмент вывода результата решения задачи имеет вид:

```
...
 WriteLn(M);
 For i:=1 To N Do
   If bb[i] Then Write(ls[i], ' ');
...
```

Сейчас мы знаем, к чему стремиться, поэтому перейдем к основной процедуре.

```
Procedure Solve;
 Var   aa: TPoint;
       i, j, r: Integer;
 Begin
  j:=1;{*Предполагаем, что самой левой верхней
         точкой множества по значению координаты X
         является первая точка.*}
  For i:=2 To N Do {*Поиск точки, заведомо
         принадлежащей выпуклой оболочке.*}
   If RealMore(A[j].x, A[i].x)
        Or (RealEq(A[i].x, A[j].x)
        And RealMore(A[i].y, A[j].y))
     Then j:=i;
       aa:=A[1];A[1]:=A[j];A[j]:=aa;
       r:=ls[1];ls[1]:=ls[j];ls[j]:=r;{*Меняем точки,
            на первом месте в массиве записана точка,
            принадлежащая выпуклой оболочке.*}
{*Вычисляем значения синусов углов, углы принадлежат
первой и четвертой четвертям, если брать их с
противоположным знаком и затем отсортировать в
порядке неубывания, то выпуклая оболочка будет
построена обходом против часовой стрелки.*}
    For i:=2 To N Do
     Begin
       aa.x:=A[i].x-A[1].x;aa.y:=A[i].y-A[1].y;
       Rd[i]:=-aa.y/Sqrt(Sqr(aa.x) + Sqr(aa.y));
     End;
    Sort(2, N);{*Обычная сортировка, например по
          Ч.Э.Р.Хоару. Следует только не забывать,
          что при перестановке ключей - Rd[i]
          необходимо переставлять и соответствующие
          элементы в  массивах ls и A (процедура не
          приводится).*}
  {*Осталось выполнить обход множества точек.*}
   If N>3 Then Rounds;{*Обход.*}
 End;
```

И наконец, последняя из рассматриваемых процедур.

```
Procedure Rounds;
 Var   lf, md, rg: TPoint;
       lfi, mdi, rgi: Integer;
```

```
        r: Real;
  Function Predd(ldi: Integer): Integer;
   Begin
     Dec(ldi);
      While Not(bb[ldi]) Do ldi:=(ldi+N-2) Mod N + 1;
      Predd:=ldi;
   End;
 Begin
  M:=N;{*Количество точек в выпуклой оболочке.*}
  lf:=A[2]; md:=A[3]; rg:=A[4];
  lfi:=2; mdi:=3; rgi:=4;
  While rgi<>2 Do
    Begin
      r:=-SquareTrian(lf, md, rg);{*Ориентированная
                              площадь треугольника.*}
      If Not(RealMore(r, 0))
        Then
          Begin
          {*Исключаем точку с номером mdi из
           выпуклой оболочки.*}
            bb[mdi]:=False;mdi:=lfi;md:=lf;
            lfi:=Predd(lfi);{*Определяем номер
                            предыдущей точки.*}
            lf:=A[lfi];
            Dec(M);
          End
        Else
          Begin {*Переходим к следующей точке.*}
            lfi:=mdi; mdi:=rgi; rgi:=rgi Mod N + 1;
            lf:=md; md:=rg; rg:=A[rgi];
          End;
    End;
 End;
```

Рассмотрим другой способ построения выпуклой оболочки — алгоритм Джарвиса. Он основан на следующем утверждении. Отрезок L, определяемый двумя точками, является ребром выпуклой оболочки тогда и только тогда, когда все другие точки множества S лежат на L или с одной стороны от него. Из этого утверждения «вытекает» следующий алгоритм. N точек определяют C_N^2, т. е. порядка $O(N^2)$ отрезков. Для каждого из этих отрезков за линейное время можно определить положение остальных $N-2$ точек относительно него. Таким образом, за время,

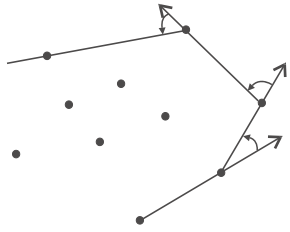


Рис. 5.24

пропорциональное $O(N^3)$, можно найти все пары точек, определяющих ребра выпуклой оболочки. Затем эти точки следует расположить в порядке последовательных вершин оболочки. Джарвис заметил, что данную идею можно улучшить, если учесть следующий факт. Если установлено, что отрезок q_1q_2 является ребром оболочки, то должно существовать другое ребро с концом в точке q_2, принадлежащее выпуклой оболочке. Уточнение этого факта приводит к алгоритму с временем работы, пропорциональным $O(N^2)$. Рисунок 5.24 служит иллюстрацией данной идеи.

Структуры данных:

```
Const MaxN=100;
Var   A: Array[1..MaxN] Of TPoint;{*Массив
          с координатами точек плоскости.*}
      N: Integer;{*Количество точек.*}
      rs: Array[1..MaxN] Of Integer;{*Номера точек,
          принадлежащих выпуклой оболочке,
          в процедуре типа Init (инициализации
          и ввода исходных данных) необходим
          оператор FillChar(rs,SizeOf(rs),0).*}
      M: Integer;{*Количество точек в выпуклой
          оболочке.*}
```

Найдем номер самой левой нижней точки. Она принадлежит выпуклой оболочке. Задача эквивалентна поиску номера минимального элемента в массиве.

```
Function GetLeft: Integer;
Var i, lf: Integer;
Begin
  lf:=1;
  For i:=2 To N Do
   If RealLess(A[i].x, A[lf].x) Or
        (RealEq(A[i].x, A[lf].x) And
         RealLess(A[lf].y, A[i].y))
     Then lf:=i;
  GetLeft:=lf;
End;
```

Основная процедура.

```
Procedure Solve;
 Var  nx: Integer;
      ls: TPoint;
 Begin
  M:=0;{*Количество точек в выпуклой оболочке.*}
  nx:=GetLeft;{*Находим самую левую и нижнюю
                точку.*}
  While (M=0) Or (nx<>rs[1]) Do
   Begin{*Пока не вернулись к первой точке.*}
    Inc(M); rs[M]:=nx;{*Очередная точка выпуклой
                       оболочки.*}
    If M>1 Then ls:=A[rs[M-1]]{*Предыдущая точка
                               выпуклой оболочки.*}
           Else
            Begin ls:=A[rs[1]]; ls.y:=ls.y-1; End;
              {*Если рассматриваемая точка является
              первой точкой выпуклой оболочки, то
              искусственно уменьшаем на единицу
              ординату у первой точки и считаем ее
              предыдущей точкой оболочки.*}
              nx:=GetNext(ls, A[rs[M]]);{*Поиск
              следующей точки выпуклой оболочки,
              параметрами функции являются координаты
              двух предыдущих точек оболочки.*}
   End;
 End;
```

Перенесли, как обычно, всю «тяжесть» алгоритма на следующую часть — функцию *GetNext* (метод последовательного уточнения, технология «сверху — вниз» в действии). Рассмотрим, как осуществляется поиск точки выпуклой оболочки, изображенной на рисунке 5.25. Требуется найти точку множества, такую, что угол, образованный лучами (*A*[*i*],*gn*) и (*gn*,*pr*), имеет минимальное значение. Если таких точек несколько, то выбирается точка, находящаяся на максимальном расстоянии от точки *gn*. Таким образом, мы опять возвращаемся к задаче поиска минимального элемента в массиве.

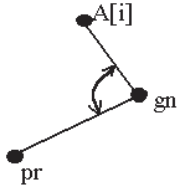


Рис. 5.25

```
Function GetNext(Const pr, gn: TPoint): Integer;
 Var  i, fn: Integer;
      mx, rsx, nw: Real;
```

```
Begin
 mx:=-10;{*Использование явных констант в «теле»
        программного кода - дурной тон, но простит
        нас читатель за эту маленькую слабость.*}
 For i:=1 To N Do
   Begin
     nw:=GetAngle(pr, gn, A[i]);{*Найдем угол.*}
     If RealLess(mx, nw)
       Then
         Begin
           fn:=i;mx:=nw; rsx:=Dist(gn, A[i]);
         End
       Else
         If RealEq(nw, mx) Then
           Begin
             nw:=Dist(gn, A[i]);
             If RealLess(rsx, nw) Then
               Begin
                 fn:=i; rsx:=nw;
               End;
           End;
   End;
 GetNext:=fn;
End;
```

Осталось уточнить функцию вычисления угла по трем точкам. Она отличается от ранее рассмотренной функции, поэтому необходимо сказать несколько слов.

Пусть L_1 и L_2 — две прямые, заданные уравнениями

$A_1 \times x + B_1 \times y + C_1 = 0$, $A_1 \times A_1 + B_1 \times B_1 > 0$,
$A_2 \times x + B_2 \times y + C_2 = 0$, $A_2 \times A_2 + B_2 \times B_2 > 0$

соответственно (см. рисунок 5.26). Угол α между прямыми L_1 и L_2 равен углу между нормальными векторами $\boldsymbol{n}_1=\{A_1,B_1\}$ и $\boldsymbol{n}_2=\{A_2,B_2\}$ этих прямых. Отсюда следует, что

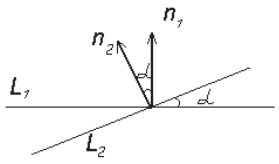


Рис. 5.26

$$\cos\alpha = \frac{(n_1, n_2)}{|n_1| \times |n_2|} = \frac{A_1 \times A_2 + B_1 \times B_2}{\sqrt{A_1^2 + B^2} \times \sqrt{A_2^2 + B_2^2}}$$

```
Function GetAngle(Const A, B, C: TPoint): Real;
 Begin
  If RealEq(Dist(C,B), 0)
```

```
    Then GetAngle:=-11
    Else GetAngle:=((B.x-A.x)*(C.x-B.x)+(B.y-A.y)*
     (C.y-B.y))/(Dist(A,B)*Dist(C,B));
End;
```

Метод «*разделяй и властвуй*» при построении выпуклой оболочки, основан, как обычно для этого типа алгоритмов, на принципе сбалансированности, суть которого в том, что вычислительная задача разбивается на подзадачи примерно одинаковой размерности. Они решаются, и затем результаты объединяются. Метод эффективен, если время, требуемое на объединение результатов, незначительно.

Суть данного алгоритма построения выпуклой оболочки. Множество S разделяется на два подмножества S_1 и S_2 примерно равной мощности. Для каждого из подмножеств строится выпуклая оболочка. Если время объединения оболочек пропорционально количеству вершин в них — $O(N)$, то временная зависимость имеет вид: $T(N) \le 2T(N/2)+O(N)$.

Итак, разбиваем множество S из N точек на два подмножества, каждое из которых будет содержать одну из двух ломаных, соединение которых даст многоугольник выпуклой оболочки (см. рисунок 5.27). Найдем точки w и r, имеющие, соответственно, наименьшую и наибольшую абсциссы. Проведем через них прямую L, тем самым получим разбиение множества точек S на два подмножества S_1 (выше или на прямой L) и S_2 (ниже прямой L). Определим точку h, для которой треугольник $<whr>$ имеет наибольшую площадь среди всех треугольников $\{<wpr>: p \in S_1\}$. Если точек имеется более одной, то выбирается та из них, для которой угол $<hwr>$ наибольший. Точка h гарантированно принадлежит выпуклой оболочке. Действительно, если через точку h провести прямую, параллельную L, то выше этой прямой не окажется ни одной точки

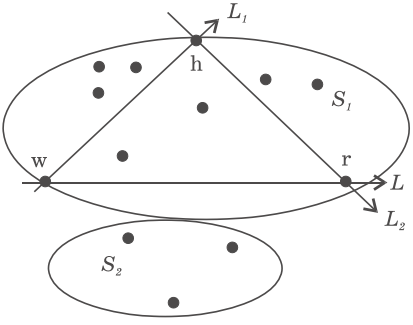


Рис. 5.27

множества S. Через точки w и h проведем прямую L_1, через точки h и r — прямую L_2. Для каждой точки множества S_1 определяется ее положение относительно этих прямых. Ни одна из точек не находится одновременно слева как от L_1, так и от L_2, кроме того, все точки, расположенные справа от обеих прямых, являются внутренними точками треугольника $<wrh>$ и поэтому могут быть удалены из дальнейшей обработки. Процесс продолжается до тех пор, пока слева от строящихся прямых типа L_1 и L_2 есть точки.

Перейдем к описанию более детального решения.

```
Const MaxN=100;
Type TSpsk = Array[0..MaxN] Of Integer;
Var    A: Array[1..MaxN] Of TPoint;
       N: Integer;
       rs: TSpsk;{*Храним номера точек, составляющих
             выпуклую оболочку, в rs[0] фиксируется
             количество точек.*}
```

Вывод результата очевиден уже в данный момент:

```
WriteLn(rs[0]);
For i:=1 To rs[0] Do Write(rs[i], ' ');
```

Опишем ряд вспомогательных функций и процедур, прежде чем разобрать алгоритм (технология «снизу — вверх»).

Нам потребуется определять взаимное расположение точки и линии на плоскости — функция *GetSign*. Значение функции равно 0, если точка находится на линии, плюс единице при нахождении в левой полуплоскости и минус единице — в правой.

```
Function GetSign(Const L: TLine; Const A: TPoint):
    Integer;
 Var rs: Real;
 Begin
  rs:=A.x*L.A + A.y*L.B + L.C;
  If RealEq(rs, 0)
    Then GetSign:=0
    Else
      If RealMore(rs, 0)
        Then GetSign:=1
        Else GetSign:=-1;
 End
```

Площадь треугольника по координатам трех вершин определяется точно так же, как и ориентированная площадь — функ-

ция *SquareTrian* (рассмотрена ранее), только берется ее абсолютное значение.

```
Function Square(Const A, B, C: TPoint): Real;
 Begin
  Square:=Abs((A.x*(B.y-C.y) + B.x*(C.y-A.y) + C.x*
   (A.y-B.y))/2);
 End;
```

Договоримся, что мы работаем с номерами точек. Координаты точек хранятся в массиве *A*. Пусть имеется некоторое множество точек, их номера в массиве *fr*. Через две заданные точки (*lf, rg*) проводится прямая линия (*L*). Требуется в массив *rs* записать номера тех точек из исходного множества, которые находятся в другой полуплоскости относительно линии *L*, чем заданная точка (*nn*). Считаем, что точки, по которым строится линия, должны быть в результирующем множестве. Решение этой задачи обеспечивает следующая процедура.

```
Procedure SelectPoints(lf, rg: Integer;
 Const nn: TPoint;
 Const fr: TSpsk;
 Var rs: TSpsk);
 Var  i, lzn, nw: Integer;
      L: TLine;
  Begin
   rs[0]:=2; rs[1]:=lf; rs[2]:=rg;
   Point2ToLine(A[lf], A[rg], L);{*Построение линии
         по двум точкам, процедура описана ранее.*}
   lzn:=GetSign(L, nn);{*Определяем знак
         полуплоскости, в которой находится точка
         с номером nn.*}
   For i:=3 To fr[0] Do
     Begin
       nw:=GetSign(L, A[fr[i]]);
       If nw*lzn=-1 Then
         Begin Inc(rs[0]);rs[rs[0]]:=fr[i]; End;
     End;
  End;
```

Пусть есть множество точек. Их номера хранятся в массиве (*ow*). Требуется найти номер той точки, которая наиболее уда-

лена от двух точек с номерами из первых двух элементов массива. Эта задача решается с помощью следующей функции. Напомним, что в нулевом элементе массива хранится количество точек множества.

```
Function MDist(Const ow: TSpsk): Integer;
 Var  max, nw: Real;
      i, rs: Integer;
 Begin
  max:=Square(A[ow[1]], A[ow[2]], A[ow[3]]);
  rs:=3;
  For i:=4 To ow[0] Do
    Begin
      nw:=Square(A[ow[1]], A[ow[2]], A[ow[i]]);
      If RealMore(nw, max) Or (RealEq(nw, max) And
        RealMore(A[ow[rs]].x, A[ow[i]].x))
        Then
          Begin max:=nw;rs:=i;End;
    End;
   MDist:=ow[rs];
 End;
```

Еще одна очевидная процедура. В исходном множестве точек находим номера самых крайних левой и правой, а также формируем первоначальный массив номеров точек. При этом на первое и второе места записываем номера найденных точек — они заведомо принадлежат выпуклой оболочке.

```
Procedure FindMinMax(Var lf, rg: Integer;
                     Var All:TSpsk);
 Var  i, sc: Integer;
 Begin
  lf:=1; rg:=1;{*Находим номера точек
         с минимальным (lf) и с максимальным (rg)
         значениями координаты x, при равных
         значениях x выбираются точки с меньшим
         значением координаты y.*}
  For i:=1 To N Do
    Begin
      If RealMore(A[i].x, A[rg].x) Or
       (RealEq(A[i].x, A[rg].x)
        And RealMore(A[rg].y, A[i].y))
          Then rg:=i;
      If RealMore(A[lf].x, A[i].x) Or
           (RealEq(A[i].x, A[lf].x)
```

```
      And RealMore(A[lf].y, A[i].y))
        Then lf:=i;
    End;
  All[0]:=N;All[1]:=lf; All[2]:=rg; sc:=2;
  For i:=1 To N Do
    If (i<>lf) And (i<>rg) Then
      Begin Inc(sc); All[sc]:=i; End;
End;
```

А сейчас одна из ключевых процедур алгоритма. Для множества точек, описываемых их номерами из массива *ow*, формируется множество точек из их выпуклой оболочки.

```
Procedure GetSpisok(Const ow: TSpsk; Var rs: TSpsk);
 Var   i, nw: Integer;
       lf, rg, rssc: TSpsk;
 Begin
  rs:=ow;
  If ow[0]>2 Then
    Begin
      nw:=MDist(ow);{*Находим максимально удаленную
        точку от точек ow[1] и ow[2].*}
      SelectPoints(ow[1], nw, A[ow[2]], ow, lf);
        {*Определяем множество точек, находящихся
        в другой полуплоскости, чем точка с номером
        ow[2], относительно прямой, проходящей
        через точки с номерами ow[1] и nw.*}
      SelectPoints(nw, ow[2], A[ow[1]], ow, rg);
        {*Определяем множество точек, находящихся
         в другой полуплоскости, чем точка
         с номером ow[1], относительно прямой,
         проходящей через точки с номерами nw
         и ow[2].*}
      GetSpisok(lf, rs);{*Рекурсивно продолжаем
         строить выпуклую оболочку для выделенного
         множества точек. Если мощность выделенного
         множества точек равна 2, то процесс
         заканчивается.*}
      GetSpisok(rg, rssc);
      For i:=2 To rssc[0] Do rs[rs[0]+i-1]:=rssc[i];
           {*Объединяем два множества точек.*}
        Inc(rs[0], rssc[0]-1);
    End;
End;
```

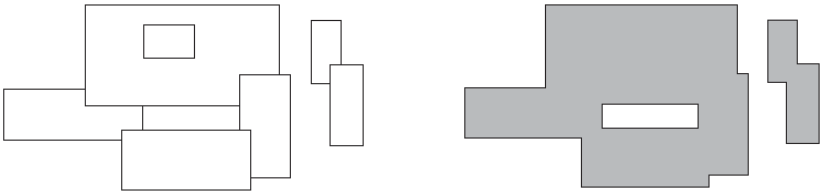
Основная процедура после проделанной работы достаточно «прозрачна».

```
Procedure Solve;
 Var   lf, rg, i: Integer;
       fr, fr1: TSpsk;
       nw: TPoint;
 Begin
  FindMinMax(lf, rg, rs);{*Находим крайние по
        значению координаты x точки исходного
        множества.*}
  nw:=A[lf];nw.y:=nw.y-10;
  SelectPoints(lf, rg, nw, rs, fr);{*B fr
        формируется список номеров точек, лежащих
        по одну сторону от прямой, проходящей через
        точки lf и rg.*}
  nw.y:=nw.y+20;
  SelectPoints(rg, lf, nw, rs, fr1);{*B fr1 - по
        другую сторону.*}
  GetSpisok(fr, rs);{*Строим выпуклую оболочку для
        множества точек с номерами из fr.*}
  GetSpisok(fr1, fr);{*Строим выпуклую оболочку
        для множества точек с номерами из fr1.*}
  For i:=2 To fr[0]-1 Do rs[rs[0]+i-1]:=fr[i];
        {*Объединяем выпуклые оболочки.*}
    Inc(rs[0], fr[0]-2);
End;
```

5.6. Задачи о прямоугольниках

1. Дано N ($0 \le N < 5000$) прямоугольников со сторонами, параллельными координатным осям. Каждый прямоугольник может быть частично или полностью перекрыт другими прямоугольниками. Вершины всех прямоугольников имеют целочисленные координаты в пределах [−10000, 10000] и каждый прямоугольник имеет положительную площадь. Периметром называется общая длина внутренних и внешних границ фигур, полученных путем наложения прямоугольников. Найти значение периметра.

На рисунке 5.28(а) показана фигура из 7 прямоугольников. Соответствующие границы полученной фигуры показаны на следующем рисунке 5.28(б).



(а) (б)

Рис. 5.28

Пример *входного файла*:
7 (количество прямоугольников)
−15 0 5 10 (границы прямоугольника — нижняя левая и верхняя правая)
−5 8 20 25
15 −4 24 14
0 −6 16 4
2 15 10 22
30 10 36 20
34 0 40 16

Ответ: 228.

Рассмотрим более простую задачу. На прямую «набросали» какое-то количество отрезков. Последние могут пересекаться, находиться один внутри другого и т. д. Требуется определить количество связных областей, образованных этими отрезками. На рисунке 5.29 даны три отрезка (2, 5), (4, 6) и (8,10). Количество связных областей две. Как их найти?

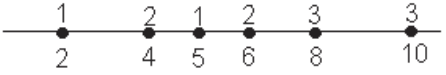


Рис. 5.29

Используем стандартный прием. Записываем в один массив координаты отрезков, в другой признаки начала (1) и конца отрезков (−1). Сортируем элементы первого массива по неубыванию, одновременно переставляя соответствующие элементы второго массива. После этого осталось считать сумму значений признаков. Если ее текущее значение равно нулю, то выделена очередная связная область.

Вернемся к исходной задаче. Очевидно, что можно считать периметр отдельно, по частям. Например, первоначально его составляющую по оси *Y*, а затем по оси *X*.

Рассмотрим идею решения на примере рисунок 5.30. Даны четыре прямоугольника. Считаем составляющую периметра по оси *Y*. Ось разбивается координатами прямоугольников на следующие участки: (0,1), (1,2), (2,3), (3,4) и (4,6). Рассматриваем каждый участок, т. е. определяем прямоугольники (точнее их координаты по *X* с признаками начала и конца интервала), которые могут дать «вклад» в периметр. Для первого участка — один прямоугольник, для второго — три, для третьего — два, для четвертого — три и для пятого — один. Выписав для каждого участка координаты *X* с соответствующими признаками, выполняем сортировку и находим количество связных областей. Так, на рисунке 5.30 для первого сечения (отмечено цифрой 1) количество связных областей равно 1, а для второго сечения — 2. После этого достаточно длину этого участка умножить на количество связных областей и удвоить результат. Получаем составляющую периметра по оси *Y* на этом участке.

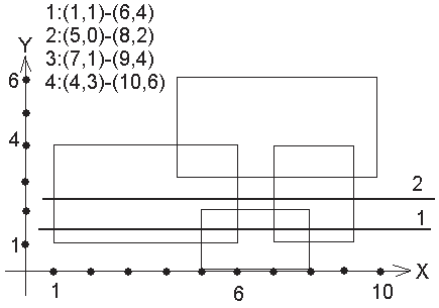


Рис. 5.30

После этих замечаний можно приступить к разбору решения. Как обычно, начинаем со структур данных.

```
Const MaxN=5001;
Type  MasPn=Array[1..MaxN*2] Of Integer;
      MasSw=Array[1..MaxN*2, 1..2] Of Integer;
         {*Для записи координат и соответствующих
         признаков начала и конца отрезков.*}
Var  N, i: Integer;
     Ax, Ay: MasPn;{*Для хранения координат
         прямоугольников.*}
```

Основная программа.

```
Begin
  Assign(Input, '…');Reset(Input);Read(N);
  For i:=1 To N Do
    Begin
      Read(Ax[i*2-1], Ay[i*2-1]);
      Read(Ax[i*2], Ay[i*2]);
    End;
  Close(Input);
  Assign(Output, '…');Rewrite(Output);
  WriteLn(GetPerim(Ax, Ay) + GetPerim(Ay, Ax));
  Close(Output);
End.
```

Вспомогательную процедуру сортировки по любому методу с временной оценкой $O(N \times logN)$ описывать не будем.

```
Procedure Sort(Var nn: MasSw; lf, rg: Integer);
 Begin
...
 End;
```

Функция *GetPerim*.

```
Function GetPerim(Const Ax, Ay: MasPn): LongInt;
 Var  i: Integer;
      sc: LongInt;
      D: MasSw;
 Begin
  sc:=0;{*Значение периметра по одному из
      направлений.*}
  FillChar(D, SizeOf(D), 0);
  For i:=1 To 2*N Do D[i, 1]:=Ax[i];{*Массив
      координат по этому направлению.*}
  Sort(D, 1, 2*N);{*Сортировка.*}
  For i:=1 To N*2-1 Do
   If D[i, 1]<>D[i+1, 1] Then{*Если координаты не
      равны, то вычисляем количество связных
      областей на этом сечении – функция GetPr,
      умножаем на длину участка и на 2.*}
    sc:=sc+GetPr(Ax, Ay, D[i, 1]) * (D[i+1, 1] -
      D[i, 1]) * 2;
  GetPerim:=sc;
 End;
```

Следующий шаг уточнения. Вычисляем количество связных областей на сечении, определяемом значением переменной *x*.

```
Function GetPr(Const Ax, Ay: MasPn; x: Integer):
                                Integer;
 Var  i, b, d, sc: Integer;
      nn: ^MasSw;{*Число прямоугольников многовато
          (до 5000), задействуем «кучу».*}
 Begin
  New(nn);
  GetRect(Ax, Ay, x, nn^, sc);{*Определяем
          прямоугольники, имеющие отношение к
          рассматриваемому сечению, их количество -
          значение переменной sc, а координаты в
          массиве nn^.*}
  If sc<>0 Then Sort(nn^, 1, sc);{*Выполняем
         сортировку с одновременной перестановкой
         признаков начала и конца отрезков.*}
  b:=0;{*Для хранения суммы значений признаков.*}
  If sc>0 Then d:=1 Else d:=0;
  For i:=1 To sc-1 Do
    Begin
      b:=b+nn^[i, 2];
      If (b=0) And (nn^[i+1, 1]<>nn^[i, 1])
        Then Inc(d);{*Если текущее значение b равно нулю
           и координаты не совпадают, то увеличиваем
           счетчик числа связных областей.*}
    End;
  GetPr:=d;
  Dispose(nn);
 End;
```

Последний шаг уточнения алгоритма — определение характеристик сечения.

```
Procedure GetRect(Ax, Ay: MasPn; x: Integer;
     Var nn: MasSw;Var sc: Integer);
 Var i: Integer;
 Begin
  sc:=0;
  For i:=1 To N Do
   If (Ax[i*2-1]<=x) And (x<Ax[i*2]) Then
     Begin
     {*Если прямоугольник имеет отношение
             к сечению, то запоминаем координаты
             по соответствующему направлению (они
             в массиве Ay) и признаки начала
             и конца отрезка.*}
```

```
      Inc(sc);
      nn[sc, 1]:=Ay[i*2-1];nn[sc+1, 1]:=Ay[i*2];
      nn[sc, 2]:=1;nn[sc+1, 2]:=-1;
      Inc(sc);
    End;
End;
```

2. На плоскости задано N ($1 \le N \le 300$) прямоугольников со сторонами, параллельными координатным осям (пример на рисунке 5.31(а)). Координаты вершин прямоугольника (x,y) — вещественные числа с точностью до двух знаков после запятой. Написать программу поиска площади фигуры (см. рисунок 5.31(б), получающейся в результате объединения прямоугольников.

Продлим все вертикальные линии до пересечения с осью X. Эти линии определяют на оси X множество интервалов, внутри каждого из которых длина сечения по оси Y будет постоянной (рисунок 5.31(в)). Поэтому для нахождения площади достаточно отсортировать точки на оси X, рассмотреть все сечения и добавить к площади <длину интервала> * <длину сечения>. Для поиска длины сечения используем метод из предыдущей задачи. Началу каждого отрезка присвоим значение признака, равное 1, а его концу — −1 и отсортируем точки по значению координаты Y. Считаем сумму значения признаков. Если она не равна нулю, то соответствующий интервал «плюсуем» к длине сечения.

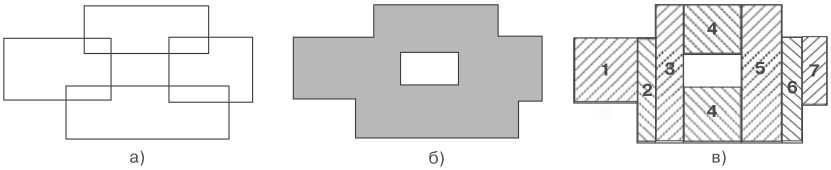


Рис. 5.31

Из структур данных приведем только те, которые требует уровень детализации объяснения.

```
Const MaxN=300;
Type TPoint=Record
            X,Y:Real
            End;
Var  PrM: Array[1..2,1..MaxN] Of TPoint;
```

```
N: LongInt;
Res: Real;
Ox, Oy: Array[1..MaxN*2] Of Real;
```

В процедуре инициализации и ввода данных координаты вершин прямоугольника записываются в массиве *PrM*, причем в *PrM*[1] (это массив) — координаты первой вершины, а в *PrM*[2] — второй. Вершина прямоугольника с меньшими значениями (*X*,*Y*) записывается в *PrM*[1]. Кроме того, значения *X* запоминаются в массиве *Ox*.

На рисунке 5.32 приведен пример записи входных данных и ответ, соответствующий этим входным данным.

Пример.

4 (количество прямоугольников)
0 0 10.1 5.2 (координаты нижнего левого и правого верхнего углов)
−3 3 5.36 7
1 6 9 15
8 3 20 8

Ответ.
195.188

Рис. 5.32

Основная процедура вычисления площади объединения прямоугольников выглядит следующим образом.

```
Procedure Solve;
 Var  i: LongInt;
      m: Real;
 Begin
  Sort(Ox, 1, N*2);{*Сортируем по неубыванию
          значения координаты X прямоугольников.
          Процедура, в силу ее очевидности, не
          приводится.*}
  m:=0;Res:=0;{*m - длина сечения, Res - значение
           площади объединения прямоугольников.*}
  For i:=1 To N*2 Do
    Begin
      If i<>1 Then Res:=Res + Abs((Ox[i]-Ox[i-1])*m);
         {*Прибавляем площадь очередного сечения.*}
```

```
      If (i=1) Or Not(Eq(Ox[i], Ox[i-1])) Then Calc
       (Ox[i], m); {*Определяем новое значение длины
                    сечения.*}
    End;
  End;
```

Уточнению подлежит процедура *Calc*.

```
  Procedure Calc(Const x: Real; Var rs: Real);
   Var i, M: LongInt;{*M – количество интервалов
       на данном сечении.*}
   Begin
    <Инициализация переменных процедуры, в частности
    M:=0>
    For i:=1 To N Do{*Формируем массив ординат для
                     данной координаты x.*}
     <Если есть пересечение прямоугольника с прямой,
параллельной оси Y и проходящей через точку x, то
занести координаты Y прямоугольника в рабочий массив,
не забыв при этом сформировать признаки начала и
конца отрезка. Количество пересечений – значение
переменной M.>;
    If M=0 Then rs:=0
          Else
     Begin
     <Сортируем границы интервалов по оси Y,
переставляя одновременно соответствующие элементы
массива признаков>;
     <Вычисляем новую длину сечения – значение
переменной rs>;
     {*Так, для примера на рисунке 5.33 длина сечения
равна 10.*}
     End;
  End;
```

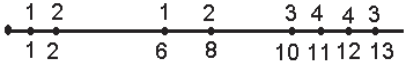


Рис. 5.33

Дальнейшее уточнение алгоритма вынесем на самостоятельную работу.

3. Найти площадь пересечения N прямоугольников со сторонами, параллельными осям координат.

Входные данные. Число N (количество прямоугольников $2 \le N \le 1000$) и N четверок действительных чисел X_{i1}, Y_{i1}, X_{i2},

Y_{i2}, задающих координаты левого нижнего и правого верхнего углов прямоугольника.

Выходные данные. Площадь пересечения (общей части) всех данных прямоугольников, либо выдать, что они не пересекаются.

На рисунке 5.34 приведен пример входных данных и соответствующий ему ответ.

Задача предельно проста. Приведем ее с целью «закрыть» тему «Прямоугольники на плоскости со сторонами, параллельными осям координат, их периметр и площади». Оказывается, что можно независимо найти область пересечения интервалов на прямой отдельно по осям X и Y. Эти интервалы определяют прямоугольник, являющийся пересечением заданных прямоугольников.

Пример.

3
0 0 10 10
5 5 15 15
−1.5 0 7 8

Ответ: 6

Рис. 5.34

```
Const MaxN=1000;
      Eps=1e-7;
Type  Segm=Record
           L, R: Real
           End;
      SegmArray=Array[1..MaxN] Of Segm;
Var   N: Integer;
      OnX, OnY: SegmArray;
      ObX, ObY: Segm;

Procedure Init;
 Begin
  <ввод данных>
 End;

Function Cross(A,B:Segm;Var New:Segm):Boolean;
      {*Устанавливаем факт пересечения двух
        отрезков A и B.*}
 Begin
  If (B.L-A.L)<Eps Then New.L:=A.L Else New.L:=B.L;
  If (A.R-B.R)<Eps Then New.R:=A.R Else New.R:=B.R;
  Cross:=(New.L-New.R)<Eps;
 End;
Function SolSegm(Const On: SegmArray;
   Var Ob: Segm): Boolean;{*Определяем область
        пересечения интервалов на прямой, если
        она есть.*}
```

```
 Var   i: Integer;
 Begin
  Ob:=On[1];
  i:=2;
  While (i<=N) And Cross(On[i], Ob, Ob) Do Inc(i);
  SolSegm:=i>N;
 End;
Begin
 Init;{*Стандартный ввод данных.*}
 Write('Ответ > ');
 If SolSegm(OnX, ObX) And SolSegm(OnY, ObY)
   Then WriteLn('Площадь ',
             Abs((ObX.L-ObX.R)*(ObY.L-ObY.R)):0:6)
   Else WriteLn(*Пересечения нет.*);
End.
```

5.7. Задачи

1. Дана прямоугольная призма, основанием которой является выпуклый N-угольник ($1 \le N \le 50$). Необходимо разделить ее на K частей ($1 \le K \le 50$) равного объема. Разрешается проводить прямые вертикальные разрезы от одной границы призмы до другой. Различные разрезы могут иметь общие точки лишь в своих концевых вершинах.

Написать программу построения требуемых $K-1$ разрезов.

Указание. В основании призмы лежит выпуклый многоугольник, в этом вся суть решения, его определенная простота. Задача решается на уровне многоугольника, лежащего в основании. Через его вершину всегда можно провести отрезок прямой и «отсечь» площадь, меньшую площади исходного многоугольника, причем оба получившихся многоугольника будут выпуклыми. Отсюда следует, что многоугольник можно «разрезать» на куски площади $S_{\text{многоугольника}}$/<число частей>, проведя все разрезы через одну вершину. Число прямых равно числу частей минус 1.

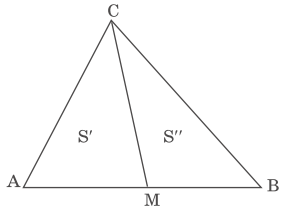


Рис. 5.35

Пусть мы нашли площадь треугольника $A_1A_NA_{N-1}$, она меньше площади отрезаемого куска. Добавляем к ней площадь треугольника $A_1A_{N-1}A_{N-2}$. Если по-прежнему площадь куска не превышена, то продолжаем процесс. В общем, мы рано или

поздно придем к случаю, когда потребуется отсечь от какого-то треугольника треугольник площадью S' (площадь второго получаемого треугольника (S'') также известна), причем одна точка разреза — вершина, а вторая «плавает» по противоположной стороне. Нам необходимо вычислить координаты точки M (см. рисунок 5.35). Высоты треугольников равны, поэтому

$$\frac{S'}{S''} = \frac{AM}{MB}.$$

После этого следует изменить выпуклый многоугольник («выкинуть» из него те точки, которые мы включили в отрезанный кусок). Легче это сделать, если начать процесс отрезания с A_1A_N, затем A_NA_{N-1}, а не A_1A_2, A_2A_3 и т. д. В этом случае не придется сдвигать элементы массива A.

2. Дан простой многоугольник со сторонами, параллельными осям координат. Три последовательно соединенных вершины многоугольника не лежат на одной прямой. Требуется определить, существует ли такая точка внутри многоугольника, из которой видимы все внутренние точки многоугольника.

Число вершин многоугольника N ($N \le 100$) и их координаты заданы в порядке обхода по часовой стрелке. Все координаты — целые. Ответом является сообщение *Yes* или *No*.

Воспользуемся следующим алгоритмом: все вертикальные отрезки поделим на две группы — восходящие и нисходящие (см. рисунок 5.36). Аналогично и для горизонтальных отрезков (идущих вправо и влево). Среди первых выберем отрезок с максимальной абсциссой (*max*), среди второй группы — с минимальной абсциссой (*min*). Назовем условие выполнимым, если $max \le min$. Искомая точка существует в том случае, если это условие выполняется как по оси X, так и по оси Y. Объяснение этому факту достаточно простое. Искомая точка находится в заштрихованной полуплоскости для каждого рассматриваемого отрезка. Если условие не выполняется, то полуплоскости не пересекаются. Приведите возможный текст решения.

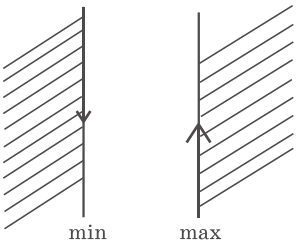


Рис. 5.36

3. *Отражение агрессии*. Для своевременного отражения возможной воздушной агрессии на город необходимо обнаруживать средства воздушного нападения противника в момент подлета к границе «опасной зоны» — на расстояние R от центра

города, который мы примем за начало координат. Имеются N комплексов ПВО, расположенных в точках с координатами (x_1, y_1), (x_2, y_2), ..., (x_N, y_N). Для каждого комплекса i ($1 \le i \le N$) известен радиус его действия R_i.

Напишите программу, которая:

- определяет, надежно ли защищен город от нападения, т. е. верно ли, что обнаружение происходит при подлете к границе «опасной зоны» с любого направления;
- в случае отрицательного ответа на первый пункт определяет диапазоны «опасных» направлений, т. е. таких направлений, с которых при подлете обнаружение не происходит. Направление задается углом (от 0° до 360°), на который нужно повернуть луч Ox против часовой стрелки, чтобы он совпал с этим направлением. Например, направление на точку $(R,0)$ соответствует числу 0, а на точку $(0,R)$ — числу 90.

В первой строке входного файла содержится натуральное число N ($1 \le N \le 50$) и вещественное число R. В последующих строках записано N троек вещественных чисел (x_i, y_i, R_i), каждая из которых задает координаты и радиус действия одного из комплексов ПВО. Числа разделяются пробелами и/или символами перевода строки. Все вещественные числа по модулю не превосходят 1000.

Первая строка выходного файла должна содержать ответ *Yes/No*. В случае отрицательного ответа вторая строка должна содержать количество диапазонов «опасных» направлений (K), а каждая из последующих K строк — начало и конец очередного из диапазонов. Диапазоны должны быть перечислены в порядке обхода против часовой стрелки.

Указание. Рассмотрим рисунок 5.37. Лучше вначале проверить, не защищает ли какая-нибудь одна ПВО весь город. Если да, то ответ очевиден, иначе определим все точки пересечений зон ПВО и зоны города. Точки типа 2 и 3 исключаем, а точки типа 1, 2, 4, 5 сортируем по значению полярного угла. Все эти точки являются защищенными, а значит, те интервалы, которые они (точки) разделяют, не объединяются. После этого остается проверить интервалы на принадлежность какой-нибудь зоне ПВО.

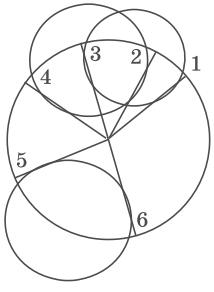


Рис. 5.37

4. На плоскости заданы треугольник и отрезок. Определить их взаимное расположение.

5. На плоскости заданы два треугольника (координатами своих вершин).

Определить взаимное расположение треугольников и найти замкнутую ломаную линию, ограничивающую область, общую для обоих треугольников (если она есть).

6. На плоскости заданы два треугольника. Используя перемещения и повороты, определить, можно ли один из треугольников вложить в другой.

7. Даны два прямоугольника со сторонами, параллельными осям координат. Определить их взаимное расположение. Для случая, когда прямоугольники пересекаются, вычислить:

- замкнутую ломаную, ограничивающую область, объединяющую оба прямоугольника;
- замкнутую ломаную, ограничивающую область, общую для обоих прямоугольников;
- площадь области, общей для обоих прямоугольников.

Снять ограничение о параллельности сторон прямоугольников осям координат. Исследовать задачу.

8. На плоскости координатами своих вершин заданы два прямоугольника. Путем их перемещений и поворотов установить, можно ли один из прямоугольников вложить в другой.

9. На плоскости задано N точек. Найти две наиболее удаленные друг от друга точки (диаметр множества).

Утверждение. Диаметр множества равен диаметру его выпуклой оболочки.

10. На плоскости задано N точек. Требуется разбить их на k групп S_1, S_2, ..., S_k таким образом, чтобы максимальный из диаметров групп был минимален.

11. На плоскости заданы N точек. Найти две из них, расстояние между которыми наименьшее.

12. На плоскости заданы N точек. Найти ближайшего «соседа» для каждой точки.

13. Из заданного множества точек на плоскости выбрать две точки так, чтобы количество точек, лежащих по сторонам от прямой, проходящей через эти две точки, различалось наименьшим образом.

14. Определить радиус и центр окружности, на которой лежит наибольшее число точек заданного на плоскости множества точек.

15. На плоскости задано множество N произвольным образом пересекающихся отрезков прямых линий. Перечислить множество всех треугольников, образованных указанными отрезками.

16. На плоскости заданы N точек. Найти минимальное количество прямых, на которых можно разместить все точки.

17. *Евклидово минимальное остовное дерево (каркас).* На плоскости заданы N точек. Построить дерево, вершинами которого являются все заданные точки, и суммарная длина всех ребер минимальна.

18. *Триангуляция.* На плоскости заданы N точек. Соединить их непересекающимися отрезками таким образом, чтобы каждая область внутри выпуклой оболочки этого множества точек являлась треугольником.

19. *Минимальное евклидово паросочетание.* На плоскости задано $2\times N$ точек. Объединить их в пары, соединив отрезками так, чтобы сумма длин отрезков была минимальна.

Примечание

Известен алгоритм со сложностью $O(N^3)$.

20. Наименьшая охватывающая окружность. На плоскости заданы N точек. Найти наименьшую окружность (по значению радиуса), охватывающую все заданные точки.

Примечание

Эта задача известна под названием *«минимаксная задача о размещении центра обслуживания»*. Требуется найти точку $t_0=(x_0,y_0)$ (центр окружности), для которой наибольшее из расстояний до точек заданного множества минимально. Точка t_0 определяется критерием $Min\{Max\{(x_i-x_0)^2+(y_i-y_0)^2\}\}$.

21. Наибольшая «пустая» окружность. На плоскости заданы N точек. Найти наибольшую окружность (см. рисунок 5.38), не содержащую внутри ни одной точки этого множества. Центр окружности должен находиться внутри выпуклой оболочки множества точек.

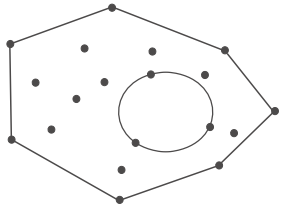


Рис. 5.38

Примечание

Известно, что как задачу о наименьшей охватывающей окружности, так и задачу о наибольшей «пустой» окружности можно решить за время, пропорциональное $O(N\times LogN)$.

22. Даны два выпуклых многоугольника Q_1 и Q_2 с N и M вершинами соответственно. Построить их пересечение.

Указание. Пересечением выпуклых многоугольников с N и M вершинами является выпуклый многоугольник, имеющий не более $N+M$ вершин.

23. Даны два выпуклых многоугольника Q_1 и Q_2 с N и M вершинами. Найти выпуклую оболочку их объединения.

Указание. Метод обработки (см. рисунок 5.39). Находим некоторую внутреннюю точку t многоугольника Q_1 (центр масс трех любых вершин Q_1). Определяем, является ли точка t внутренней точкой многоугольника Q_2 (за время, пропорциональное $O(N)$). Если точка t является внутренней точкой Q_2, то вершины как Q_1, так и Q_2 упорядочены в соответствии со значением полярного угла относительно точки t. За время, пропорциональное $O(N)$, получаем упорядоченный список вершин как Q_1, так и Q_2 путем слияния списков вершин этих многоугольников. К полученному списку точек применяем метод обхода Грэхема. Если точка t не является внутренней точкой Q_2, то многоугольник Q_2 лежит в клине с углом, меньшим или равным π. Этот клин определяем двумя вершинами u и v многоугольника Q_2, которые находим за один обход вершин многоугольника Q_2. Вершины u и v разбивают границу Q_2 на две цепочки вершин, монотонные относительно изменения полярного угла с началом в точке t. При движении по одной цепочке угол увеличивается, при движении по другой — уменьшается. Одну из этих цепочек удаляем (ее точки являются внутренними относительно строящейся оболочки). Другая цепочка и граница Q_1 являются упорядоченными списками. Соединяем их в один список вершин, упорядоченный по полярному углу относительно точки t, и выполняем обход.

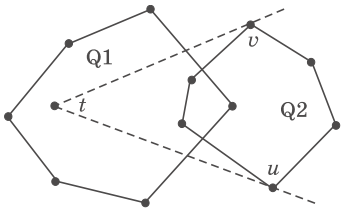


Рис. 5.39

24. Определить принадлежность точки p выпуклому N-угольнику Q. (см. рисунок 5.40)

Указание. Методом двоичного поиска находим «клин». Точка p лежит между лучами, определяемыми q_i и q_{i+1}, тогда и то-

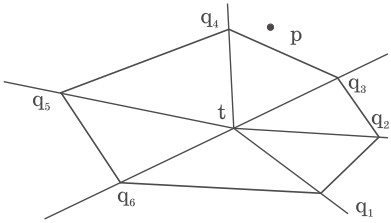


Рис. 5.40

лько тогда, когда угол (ptq_{i+1}) положительный, а угол (ptq_i) отрицательный. Сравниваем p с единственным ребром. Точка p внутренняя тогда и только тогда, когда угол ($q_iq_{i+1}p$) отрицательный.

25. Разработать функции:
- построения окружности по трем точкам;
- нахождения пересечения прямой и окружности;
- нахождения точек пересечения двух окружностей.

26. Построить выпуклую оболочку для множества точек на плоскости без использования операций с вещественным типом данных.

Указание [2]. Строим выпуклую оболочку в порядке обхода против часовой стрелки. Находим самую правую нижнюю точку (x_0,y_0). Она принадлежит выпуклой оболочке. Следующей является точка (x_1,y_1), для которой угол между осью X и вектором (x_0,y_0) – (x_1,y_1) минимален. Если таких точек несколько, то выбирается точка, расстояние до которой максимально.

Суть идеи заключается в том, чтобы угол не вычислять, а вычислять знак выражения $(x_{i+1}-x_i)*(y-y_i) - (y_{i+1}-y_i)*(x-x_i)$, где ($x_i$,$y_i$) — очередная точка, вошедшая в выпуклую оболочку, (x_{i+1},y_{i+1}) — следующая точка выпуклой оболочки, а (x,y) — координаты любой точки, не вошедшей в выпуклую оболочку. Выражение отрицательно, и это является «ключом» решения задачи.

27. На плоскости заданы N точек. Построить замкнутую ломаную без самопересечений и самокасаний, координатами которой должны стать все заданные точки.

Указание. Построить часть выпуклой оболочки между самой левой и самой правой (по координате X) точками. Оставшиеся точки отсортировать по координате X и затем последовательно соединить отрезками.

6. Избранные олимпиадные задачи по программированию

1. Дано целое неотрицательное число N (N<32767). При записи N в двоичной системе счисления получается последовательность из 1 и 0. Например, при N=19 получаем $19_{10} = 1\cdot2^4+0\cdot2^3+0\cdot2^2+1\cdot2^1+1\cdot2^0$, т. е. в двоичной системе число запишется как 10011_2. При циклическом сдвиге вправо получаются другие числа. Найти максимальное среди этих чисел. Так, для числа 19 список последовательностей показан на рисунке 6.1, а результатом является число $1\cdot2^4+1\cdot2^3+1\cdot2^2+0\cdot2^1+0\cdot2^0 = 28$.

10011
11001
11100
01110
00111
10011

Рис. 6.1

Указание. Находим старший значащий разряд в двоичном представлении числа N, ибо бессмысленно, например, в числе 000000001010 выполнять лишние сдвиги, соответствующие первым нулям. После того, как значение числа сдвигов найдено, последовательно изменяем значение N и находим его максимум. Задача становится более интересной, если снять ограничение по N (N>32767). Как минимум не избежать длинной арифметики и моделирования выполнения операций сдвига на один двоичный разряд.

```
Procedure Solve;
 Var i, j: Integer;
 Begin
  Res:=N; i:=0;
  While ((N shr i)>1) Do Inc(i);{*Определяем место
          первой единицы справа в двоичном
          представлении числа N.*}
  j:=i;{*Число сдвигов*}
  While j<>0 Do
    Begin
      N:=(N shr 1) + (N And 1) shl i;{*Сдвигаем вправо
          на один разряд и приписываем старшую
          единицу.*}
      If N>Res Then Res:=N;
      Dec(j);
    End;
 End;
```

2. «*Кратеры на Луне*». Время от времени пролетающие вблизи нашего спутника Луны астероиды захватываются ее гравитационным полем и, будучи ничем не задерживаемы, врезаются с огромной скоростью в лунную поверхность, оставляя в память о себе порядочных размеров кратеры приблизительно круглой формы.

Требуется найти максимально длинную цепочку вложенных друг в друга кратеров.

Формат входных данных. Первая строка входного файла (см. рисунок 6.2) содержит целое число N — количество кратеров, отмеченных на карте ($1 \le N \le 500$). Следующие N строк содержат описания кратеров с номерами от 1 до N. Описание каждого кратера занимает отдельную строку и состоит из трех целых чисел, принадлежащих диапазону [−32768, 32767] и разделенных пробелами. Первые два числа представляют собой декартовы координаты его центра, а третье — радиус. Все кратеры различны.

Формат выходных данных. Первая строка выходного файла (см. рисунок 6.2) должна содержать длину искомой цепочки кратеров, вторая — номера кратеров из этой цепочки, начиная с меньшего кратера и кончая самым большим. Номера кратеров должны быть разделены пробелами. Если существует несколько длиннейших цепочек, нужно вывести любую из них.

Пример.

d.in	*d.out*
4 0 0 30 −15 15 20 15 10 5 10 10 10	3 3 4 1

Рис. 6.2

Указание. Ответить на вопрос о принадлежности одного кратера другому достаточно просто. Пусть координаты центров кратеров и их радиусы хранятся в массивах. Для решения задачи будем использовать данные:

```
Const MaxN=500;
Var Ax, Ay, Ar: Array[1..MaxN] Of LongInt;)
```

В каком случае кратер с номером i вложен в кратер с номером j? Если точки с координатами

$Ax[i]+Ar[i]$, $Ay[i]$,
$Ax[i]-Ar[i]$, $Ay[i]$,
$Ax[i]$, $Ay[i]+Ar[i]$,
$Ax[i]$, $Ay[i]-Ar[i]$

принадлежат кратеру с номером j, то и весь кратер с номером i находится в кратере с номером j. Точка находится внутри окружности (кратера) в том случае, если расстояние между точкой и центром окружности меньше радиуса.

Рассмотрим еще пример, изображенный на рисунке 6.3. Пусть в массивах *Sc* и *Ls*(*Array*[1..*N*] *Of Integer;*) каким-то образом оказались следующие данные: *Sc* — (3, 0, 1, 0, 1, 2) и
Ls — (6, 0, 2, 0, 4, 5).

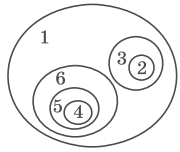


Рис. 6.3

Найти максимальный элемент *mx* в массиве *Sc* и его номер *k* несложно. Если вывести значение найденного максимума *WriteLn*(*mx*+1) и вызвать процедуру *SayRes*(*k*) (в нашем случае будет *SayRes*(1))

```
Procedure SayRes(k: Integer);
 Begin
  If Ls[k]<>0 Then SayRes(Ls[k]);
  Write(k, ' ');
 End;
```

то получим:

4
4 5 6 1,

что и является ответом задачи. Итак, осталось понять, как формировать значения элементов массивов *Sc* и *Ls*.

3. На клеточном поле задано *N* точек (в узлах клеточного поля).

Требуется найти точку, до которой суммарное расстояние от заданного множества точек минимально. Расстояние считается при прохождении по границам клеток.

Пример.
4 01
0 0
0 0
1 1
2 2

Рис. 6.4

Формат входных данных (см. рисунок 6.4). Первая строка входного файла содержит число *N* — количество точек ($1 \le N \le 500$). Каждая из последующих *N* строк содержит координаты точки — два целых числа из диапазона [−32767, 32767], разделенных пробелом.

Формат выходных данных (см. рисунок 6.4). Ваша программа должна вывести в выходной файл координаты точки — два целых числа, записанных через пробел.

Указание. Зачем нам плоскость? Почему задачу нельзя решить отдельно для каждого измерения?

Результат решения на плоскости — это результат решения на двух прямых, т. е. нашли точку на *X*, нашли точку на *Y*, тем самым получили координаты точки на плоскости. Решаем задачу на прямой линии. Пусть не *N* точек, а две. Очевидно, что точку следует размещать в том месте, где находится одна из точек. А если три точки? Например, (1, 5, 1000). В этом случае ответом является 5.

4. «Печать буклета». При печати документа обычно первая страница печатается первой, вторая — второй, третья — третьей и так далее до конца. Но иногда, при создании буклета, порядок печати должен быть другим. При печати буклета на одном листе печатаются четыре страницы: две — на лицевой стороне, и две — на обратной. Когда вы сложите все листы по порядку и согнете их пополам, страницы будут идти в правильном порядке, как у обычной книги. Например, четырехстраничный буклет (см. рисунок 6.5) должен быть напечатан на одном листе бумаги: лицевая сторона должна содержать сначала страницу 4, потом — 1, обратная — 2 и 3.

Рис. 6.5

Если в буклете число страниц не кратно четырем, то в конце можно добавить несколько пустых страниц, но так, чтобы количество листов бумаги при этом было минимально возможным.

Ваша задача — написать программу, которая считывает из входного файла количество страниц в буклете и генерирует порядок его печати.

Входные данные. Входной файл (*a.in*, см. рисунок 6.6) содержит количество страниц в буклете — натуральное число, не превышающее 500.

Выходные данные. В выходной файл (*a.out*, см. рисунок 6.6) необходимо выдать порядок печати данного буклета — последовательность команд, каждая из которых располагается на отдельной строке и состоит из четырех чисел. Числа разделяются пробелом и обозначают следующее:

- номер листа, на котором происходит печать;
- сторону: 1 — если печать происходит на лицевой стороне, и 2 — если на обратной стороне.

Два оставшихся числа — номера страниц буклета, которые должны быть напечатаны. Пустая страница задается числом 0. Если целая сторона должна быть оставлена пустой, команду для ее печати выводить не обязательно.

Указание. Необходимо дополнить число страниц до значения, кратного четырем, а затем аккуратно выводить информацию по лицевым и обратным сторонам листа буклета. На лицевых сторонах листов

Пример.

a.in

1

a.out

1 1 0 1

---

a.in

4

a.out

1 1 4 1
1 2 2 3

---

a.in

14

a.out

1 1 0 1
1 2 2 0
2 1 14 3
2 2 4 13
3 1 12 5
3 2 6 11
4 1 10 7
4 2 8 9

Рис. 6.6

буклета правая страница должна быть нечетной, а на обратных — четной.

Фрагмент решения.

```
Var n:Integer;
Function Out(q:Integer):Integer;
 Begin
  If q<=n Then Out:=q Else Out:=0;
 End;
Procedure Solve;
 Var i,nn:Integer;
 Begin
  If n=1 Then WriteLn('1 1 0 1');
         Else
    Begin
      nn:=n;
      nn:=nn+(4-nn mod 4);{*Увеличиваем количество
            страниц до значения, кратного 4.*}
      For i:=1 To nn Div 2 Do
        Begin
         If i Mod 2=1
           Then WriteLn((i+1) Div 2,' 1 ',
             Out(nn+1-i),' ', Out(i))
             {*Выводим лицевые стороны.*}
           Else WriteLn(i Div 2,' 2 ',Out(i),' ',Out(nn+1-i));
             {*Выводим обратные стороны.*}
        End;
    End;
 End;
```

5. В стране Гексландии каждое графство имеет форму правильного шестиугольника, а сама страна имеет форму большого шестиугольника, каждая сторона которого состоит из N маленьких шестиугольников (на рисунке 6.7 N=5). Графства объединяются в конфедерации. Каждая конфедерация раз в год выбирает себе покровителя — одного из 200 жрецов. Этот ритуал называется Великими перевыборами жрецов и выглядит так: конфедерации одновременно подают заявления (одно от конфедерации) в Совет жрецов о том, кого они хотели бы видеть своим покровителем (если заявление не подано, то считают, что конфедерация хочет оставить себе того же покровителя). После этого все заявки удов-

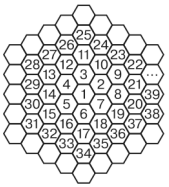


Рис. 6.7

летворяются. Если несколько конфедераций выбирают одного и того же жреца, то они навсегда объединяются в одну. Таким образом, каждый жрец всегда является покровителем не более чем одной конфедерации.

Требуется написать программу, позволяющую Совету жрецов выяснить состав каждой конфедерации после Великих перевыборов.

В Совете все графства занумерованы (начиная с 1) с центрального графства по спирали (см. рисунок). Все жрецы занумерованы числами от 1 до 200 (некоторые из них сейчас могут не быть ничьими покровителями).

Входные данные. Во входном файле (*b.in*, см. рисунорк 6.8) записано число N — размер страны ($1 \le N \le 128$) — и далее для каждого графства записан номер жреца-покровителя конфедерации, в которую оно входит (графства считаются по порядку их номеров). Затем указаны заявления от конфедераций в виде пар чисел: первое число — номер очередного жреца-покровителя, второе — номер желаемого жреца-покровителя. Все числа во входном файле разделяются пробелами и (или) символами перевода строки.

Пример.

b.in

```
2
1 1 5 3 1 5 1
5 1
1 3
```

b.out

```
3 3 1 3 3 1 3
```

Рис. 6.8

Выходные данные. В выходной файл (*b.out*, см. рисунок 6.8) вывести для каждого графства одно число — номер его жреца-покровителя после Великих перевыборов. Сначала — для первого графства, затем — для второго и т. д.

Указание. Задача на принцип косвенной адресации. Использовать два массива (начальный и конечный) для хранения номеров жрецов для графств нельзя — оперативной памяти явно не хватит. Обозначим через $f(N)$ количество графств в стране со стороной из N шестиугольников. Тогда $f(1)=1$. Заметим, что при увеличении размера страны добавляется «внешний пояс» графств, их количество — $6\times(N-1)$. Итак, $f(N)=f(N-1)+6\times(N-1)$, или $1+3\times(N-1)\times N$. При N, равном 128, получаем 48769 графств.

```
Var a: Array[1..200] Of Word;
      b : Array[1..50000] Of Byte;
      i,n,k,t : Word;
      f : Text;
Begin
 Assign(f,'b.in');Reset(f);
 Read(f,n);
 n:=1+3*(n-1)*n;
```

```
  For i := 1 To 200 Do a[i]:=i;
  For i := 1 To n Do Read(f,b[i]);
  While Not SeekEof(f) Do
    Begin
     Read(f,k,t);a[k]:=t;
    End;
  Close(f);
  Assign(f,'b.out');
  Rewrite(f);
  For i := 1 To n Write(f,a[b[i]]);
  {*Косвенная адресация.*}
  Close(f);
End.
```

А что, если N больше 128, например, равно 1000? Количество графств приближается к 3 миллионам. Есть ли выход? Оказывается, есть. Массив для хранения номеров жрецов для графств вообще не требуется.

6. «*Игра в слова*». Надеемся, что вам знакома следующая игра. Выбирается слово, и из его букв составляются другие осмысленные слова, при этом каждая из букв исходного слова может быть использована не более такого количества раз, которое она встречается в исходном слове.

Напишите программу, помогающую играть в эту игру.

Входные данные. В первой строке входного файла (*c.in*, см. рисунок 6.9) записано выбранное для игры слово. В последующих строках задан «словарь» — множество слов, которые мы считаем осмысленными. Их количество не превышает 10000. Слово — это последовательность не более чем 255 маленьких латинских букв. Каждое слово записано в отдельной строке. Заканчивается словарь словом "ZZZZZZ" (состоящим из заглавных букв), которое мы не будем считать осмысленным. Слова в словаре, как это ни странно, не обязательно располагаются в алфавитном порядке.

Выходные данные. В выходной файл (*c.out*, см. рисунок 6.9) необходимо выдать список слов, которые можно получить из исходного

Пример.

c.in

```
soundblaster
sound
sound
blaster
soundblaster
master
last
task
sos
test
bonus
done
ZZZZZZ
```

c.out

```
sound
sound
blaster
soundblaster
last
sos
bonus
done
ZZZZZZ
```

Рис. 6.9

слова. Слова должны быть выданы в том же порядке, в котором они встречаются в словаре. Каждое слово должно быть записано в отдельной строке. Список слов должен заканчиваться все тем же неосмысленным словом «ZZZZZZ».

Указание. Требуется проверить: можно ли из букв одного слова получить другое слово. «Запутанные» решения основаны на анализе соответствия букв исходных слов. При простом решении следует сформировать маски — количество букв (*a, b, c,..., z*) в каждом из слов. Если такие маски есть, то задача сводится к проверке вхождения маски слова из словаря в маску первого слова.

7. «*Делители*». Даны два положительных целых числа A и B. Определить возможность их разложения на N положительных делителей $a_1, a_2, ..., a_N, b_1, b_2, ..., b_N$ таких, что произведения этих делителей совпадают. Т. е. требуется выполнение следующих условий: $a_1 \times a_2 \times ... \times a_N = A$, $b_1 \times b_2 \times ... \times b_N = B$ и $a_1 \times b_1 = a_2 \times b_2 = ... = a_N \times b_N$, где $a_1, a_2, ..., a_N$ — делители числа A, а $b_1, b_2, ..., b_N$ — делители числа B. Число 1 в этой задаче считается делителем.

Входные данные. В единственной строке входного файла (*d.in*, см. рисунок 6.10) через пробел записаны числа A, B и N. Произведение чисел A и B не превышает значения 2147483647, а значение N — 31.

Выходные данные. В первую строку выходного файла (*d.out*, см. рисунок 6.10) через пробел записываются числа $a_1, a_2, ..., a_N$, а во вторую — $b_1, b_2, ..., b_N$ (достаточно вывести одно из решений). Если решение отсутствует, то в выходной файл записывается сообщение «*no*».

Пример.

d.in

```
4 16 2
```

d.out

```
2 2
4 4
```

Рис. 6.10

Указание. Произведение делителей чисел A и B совпадает и равно, обозначим его как h. Нахождение h сводится к вычислению

$$h := Round(Exp(ln((A*1.0)*B)/N)).$$

Пусть мы нашли число j такое, что h и A делятся нацело на j, а B нацело делится на h *Div* j. В этом случае A следует разделить на j, а B — на h *Div* j. После этого процесс поиска делителей следует продолжить. Если мы найдем N таких чисел j, то задача решена.

Пример. Если A равно 320, B — 2278125 и N — 6, то после вычислений получим h, равное 30, а в результирующих массивах будет записано соответственно (2, 2, 2, 2, 2, 10) и (15, 15, 15, 15, 15, 3).

8. «*Мотоциклисты*». Среди мотоциклистов г. Кирова популярно следующее соревнование: на одной из улиц города выбирается прямой участок, на котором установлено несколько ($1 \le N \le 100$) светофоров, имеющих только красный и зеленый цвета. Мотоциклист проезжает выбранный участок на постоянной скорости, причем нарушать правила дорожного движения, т. е. проезжать перекресток на красный свет, категорически запрещается, кроме случаев, когда мотоциклист пересекает перекресток в момент переключения светофоров. Мотоциклист, изменивший скорость движения, либо проехавший на красный свет, либо выбравший скорость, меньшую 5 м/с, дисквалифицируется. Побеждает мотоциклист, проехавший трассу за минимальное время.

Требуется найти оптимальную скорость движения по трассе.

Входные данные. В первой строке входного файла (*m.in*, см. рисунок 6.11) записано N — число светофоров. В следующей строке $N+1$ чисел: первое — расстояние от точки старта до первого светофора, второе — расстояние между первым и вторым светофорами, ... последнее число — расстояние от последнего светофора до точки финиша (расстояния — числа из интервала от 1 до 100000). В третьей строке записаны N чисел — времена работы в красном и зеленом режимах (оно одинаково) для каждого светофора (числа из интервала от 1 до 300).

Пример.

m.in

```
3
200 100 200 200
2 1 9
```

m.out

```
55.5556
```

Рис. 6.11

Выходные данные. В выходной файл (*m.out*, см. рисунок 6.11) записывается одно число (с точностью до четырех знаков) — искомая скорость или сообщение «*no*», если решения нет.

Указание. На рисунке 6.12 по горизонтальной оси откладываем расстояния, а по вертикальной — время. Времена работы каждого светофора в красном режиме выделены на соответствующих расстояниях «черным отрезком». Каждой скорости движения мотоциклиста соответствует наклонная прямая. Для решения задачи необходимо провести прямую из начала координат с наименьшим угловым коэффициентом, проходящую через «светлые» участки рисунка 6.12. Обратим внимание на то, что «лучшая» прямая касается хотя бы одного отрезка в верхней точке.

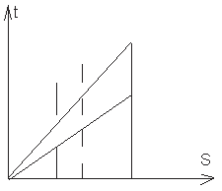


Рис. 6.12

Действительно, если она не касается ни одного отрезка в верхней точке, то ее можно сдвинуть («повернуть») вниз, уменьшая при этом угол, т. е. увеличивая скорость, до соприкосновения с первой возможной точкой. Из вышесказанного следует решение: необходимо проверить все возможные «верхние» точки «темных отрезков» на предмет их принадлежности искомым прямым и выбрать наилучшую.

9. «*Троллейбусы*». Троллейбусы одного маршрута проходят через остановку каждые k ($1 \le k \le 500$) минут. Известны времена прихода N ($1 \le N \le 100000$) жителей г. Кирова на остановку (N, k — целые числа). Если человек приходит на остановку в момент прихода троллейбуса, то он успевает войти в этот троллейбус.

Необходимо написать программу определения времени прибытия первого троллейбуса на остановку (это число от 0 до k-1), такого, чтобы:

- суммарное время ожидания троллейбуса для всех граждан было минимально;
- максимальное из времен ожидания троллейбуса было минимально.

Входные данные. В первой строке входного файла (*t.in*, см. рисунок 6.13) записано число k, во второй — N, а затем N строк со временем прихода жителей на остановку (числа от 0 до 100000).

Выходные данные. В выходном файле (*t.out*, см. рисунок 6.13) записываются два числа, каждое в своей строке, являющиеся ответами на первый и второй вопросы задачи соответственно. Если вариантов решения задачи несколько, то выдать один из них. При решении только одного пункта задачи строка, соответствующая другому пункту, остается пустой.

Указание. Достаточно подсчитать время прихода жителей по модулю k. После этого в каждый момент времени от 0 до k–1 имеем количество пришедших жителей (см. рисунок 6.14). Для ре-

Пример.

t.in

```
100
5
0
210
99
551
99
```

t.out

```
10
```

Рис. 6.13

Рис. 6.14

шения задачи достаточно проверить время прихода первого троллейбуса от 0 до $k-1$ и выбрать из них наилучшее для первого и для второго пунктов.

Так, например, алгоритм решения первого пункта задачи выглядит следующим образом.

```
Procedure SolvePA;
 Var i, j: LongInt;
     sc, Min: LongInt;
 Begin
  Min:=MaxLongInt;
  For i:=0 To k-1 Do
    Begin{*Время прихода первого троллейбуса.*}
      sc:=0;{*Находим сумму времени ожидания
             троллейбусов для всех жителей.*}
      For j:=0 To k-1 Do sc:=sc + ((k - j + i) Mod k) * A[j];
      If sc<Min Then
         Begin ResA:=i;Min:=sc;End;
         {*Сравниваем.*}
    End;
 End;
```

10. «*Посылки*». Весь воскресный день родители решили посвятить отправке посылок голодному в г. Москве сыну-студенту. Для этого родители привозят посылки на вокзал и отправляют их на любом идущем в г. Москву поезде. А сын в г. Москве приезжает к приходу этого поезда на вокзал, получает посылку и отвозит ее в общежитие. При этом родители не могут сразу принести на вокзал более одной посылки (так как очень тяжело!). Студент также не может носить по несколько посылок сразу (он голодный!). Время, необходимое родителям, чтобы добраться от дома до вокзала, равно T (оно равно времени переезда от вокзала до дома). В г. Москве, как известно, поезда приходят на разные вокзалы. Для каждого вокзала известно время, необходимое студенту на дорогу от общежития до этого вокзала (оно равно времени на дорогу от этого вокзала до общежития). Дано расписание движения поездов из г. Кирова в г. Москву. Договоримся, что поезда в пути не обгоняют друг друга (т. е. поезд, который раньше другого отправляется из г. Кирова, прибывает в г. Москву также раньше).

Написать программу определения максимального количества посылок, которые родители смогут послать за воскресный день голодному студенту (он все их должен встретить и отвезти в общежитие).

Входные данные. Во входном файле (*p.in*, см. рисунок 6.15) заданы в следующем порядке целые числа:

- T — время на дорогу родителей от дома до вокзала в г. Кирове;
- M — количество вокзалов в г. Москве ($0<M<20$);
- M строк, в каждой из которых записано одно число — время переезда от соответствующего вокзала до общежития;
- N — количество поездов ($0<N<1000$);
- N строк, каждая из которых содержит 3 числа: время отправления поезда из г. Кирова, время в пути, номер вокзала в г. Москве, на который прибывает данный поезд.

Пример.

p.in

```
70
2
40
50
4
60 1200 2
200 1120 1
1200 1440 2
```

p.out

Рис. 6.15

Примечание

Список поездов дан в отсортированном по времени отправления порядке. Время в задаче — абстрактное число из интервала от 0 до 5000. В одну единицу времени отправляется не более одного поезда. Все данные — целые числа.

Выходные данные. В выходном файле (*p.out*, см. рисунок 6.15) необходимо вывести одно число — максимальное количество посылок, которые родители смогут отправить за воскресенье.

Указание. Предположим, что родители отправили посылку с первым поездом. Принимаем решение по второму поезду. Возможны два варианта. Если они успевают съездить за второй посылкой, то они ее отправляют с поездом, и количество отправленных посылок к этому моменту времени будет равно двум, а если нет, то первую посылку они могут отправить не с первым, а со вторым поездом. Продолжим последовательно этот процесс. Пусть для всех поездов с номерами i от 1 до $k-1$ найдено максимальное количество посылок, которые успеют отправить родители к моменту отправления поезда i. Решаем задачу для поезда с номером k (рисунок 6.16), предполагая, что с ним также отправляется посылка. Для нахождения этого числа (посылок) достаточно рассмотреть все предыдущие поезда, с которыми можно было отправить предпоследнюю посылку и успеть отправить посылку на поезде с номером k. Естественно, что из этих чисел выбирается максимальное. Схема динамического программирования в действии.

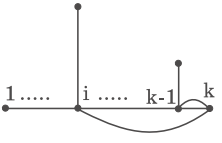


Рис. 6.16

Рассмотрим основную процедуру — она проста, если не вдаваться в детали, что и не требуется делать на данном этапе решения.

```
Procedure Solve;
 Var i, j: Integer;
 Begin
  ResP:=0;{*Результат.*}
  For i:=1 To N Do
    Begin {*Решаем задачу для поезда
          с номером i.*}
     Res[i]:=1;{*Мы всегда можем отправить первую
          посылку с этим поездом, "проспав" все
          предыдущие.*}
     For j:=1 To i-1 Do{*Согласуем отправку посылки
          с поездом i с отправками посылок на
          предыдущих поездах.*}
      If IfPossib(j,i) Then Res[i]:=Max(Res[j]+1,Res[i]);
          {*Если между поездами с номерами i и j
          родители успевают съездить домой, студент
          - в общежитие, то сравниваем Res[j]+1 и
          Res[i].*}
     ResP:=Max(Res[i], ResP);{*Может быть результат,
          полученный для поезда с номером i, Res[i]
          являться ответом задачи, например, в том
          случае, когда все следующие поезда
          отправляются через каждую секунду.*}
    End
 End;
```

Итак, все технические сложности задачи сконцентрированы в функции *IfPossib* проверки совместимости отправки посылок поездами с номерами i и j.

11. «*Закраска прямой*». Интервал прямой с целочисленными координатами $[a,b)$ содержит левую границу — точку a, и не содержит правую границу — точку b. Итак, интервал от 0 до 1.000.000.000 выкрасили в белый цвет. Затем было выполнено N ($1<=N<=100$) операций перекрашивания. При каждой операции цвета в интервале, границы которого задаются, меняются на противоположный (белый на черный, черный на белый).

Написать программу поиска самого длинного интервала белого цвета после заданной последовательности операций перекрашивания.

Входные данные. Входной файл (*z.in*, см. рисунок 6.17) содержит в первой строке число N и затем N строк с границами интервалов (числа в диапазоне *LongInt*).

Выходные данные. В выходной файл (*z.out*, см. рисунок 6.17) выводится одно число — длина самого большого белого интервала.

Пример.

z.in

```
4
20 50
10 35
40 90
100 1000000000
```

z.out

```
15
```

Рис. 6.17

Указание. Решение с отслеживанием всех перекрашиваний достаточно трудоемко. Порисуем чуть-чуть. На рисунке изображен один интервал. Вверху указан номер границы, внизу — цвет интервала. Значения границ не изображены. Границы с номерами 0 и 3 соответствуют предельным значениям.

Пусть есть два интервала. Различные варианты размещения интервалов показаны на рисунке 6.18. Кружками большего размера отмечены границы второго интервала. Можно продолжить рисование: много отрезков, интервалы, начинающиеся в одной точке и т. д., но эти рисунки будут только подтверждать идею. Если отсортировать все границы в порядке возрастания, то белые участки после всех перекрашиваний начинаются с четных номеров границ.

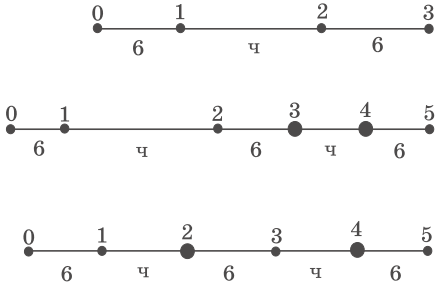


Рис. 6.18

12. «*Отрезки*». На прямой линии расположено N ($1 \le N \le 500$) отрезков. Из них нужно выбрать наибольшее количество попарно не пересекающихся. Отрезки пересекаются, если они имеют хотя бы одну общую точку.

Входные данные. В первой строке файла (*o.in*, см. рисунок 6.19) записано число N — количество отрезков. В каждой из последующих N строк заданы координаты отрезка.

Выходные данные. В выходной файл (*o.out*, см. рисунок 6.19) записываются координаты концов отрезков (целые числа из диапазона *Integer*). Каждый отрезок в отдельной строке.

Пример.

o.in

```
3
2 5
1 3
4 6
```

o.out

```
1 3
4 6
```

Рис. 6.19

Указание. Попытки решения задачи переборными методами не имеют успеха. Сгенерировать все возможные подмножества множества из 500 элементов при любых эвристических приемах сокращения перебора не реально, поэтому следовало сразу искать другое решение. Задача относится к классу задач, решаемых с помощью «жадных» алгоритмов. *Суть*: сортируем отрезки по значениям правых концов, а затем последовательно идем по отсортированным отрезкам и берем в качестве следующего отрезка такой первый отрезок, у которого правая граница больше левой границы текущего отрезка.

Основной фрагмент реализации имеет вид:

```
Const NMax=500;
Var   A:Array[0..NMax,1..2] Of LongInt;
Procedure Solve;
 Var i:Integer;
       t:LongInt;
 Begin
  t:=-MaxLongInt;
  For i:=1 To N Do
   If A[i,1]>t Then
     Begin
       Writeln(A[i,1],' ',A[i,2]); t:=A[i,2];
     End;{*Если левая граница больше значения t, то
          включаем отрезок в решение и присваиваем
          переменной t значение правой границы
          отрезка.*}
End;
```

Обоснуйте правильность его работы.

13. «*Поезда*». Имеется ветка метро, станции которой пронумерованы по порядку числами 1 (центр), 2, …, N ($2 \le N \le 30$). Из центра (станция с номером 1) и самой дальней станции (номер N) одновременно отправляются два поезда.

Требуется выяснить, где и когда произойдет встреча, если движение поездов удовлетворяет следующим условиям:

- на каждой из станций, кроме первой и последней, поезда стоят M часов;

- поезда разгоняются и тормозят мгновенно;
- во время движения поезда имеют одну и ту же скорость V;
- в случае, если поезда встречаются на станции, выдать самое раннее время встречи.

Входные данные. В первой строке файла (*p.in*, см. рисунок 6.20) задано число V — скорость поезда (в км/час). Во второй строке задано число M. В третьей строке задано число N — количество станций. В четвертой строке задан набор чисел d_2,..., d_n, где число d_i является расстоянием от станции с номером (i–1) до станции с номером i (в километрах). Все числа, кроме N, вещественные.

Выходные данные. В выходной файл (*p.out*, см. рисунок 6.20) требуется записать время встречи (в часах, с одной цифрой после точки) и расстояние от места встречи до центра (с одной цифрой после точки).

Пример.

p.in

```
1
0.5
5
1 1 1 1
```

p.out

```
2.5
2.0
```

Рис. 6.20

Указание. Задача не требует знаний специальных алгоритмов. Можно решать путем моделирования движений того и другого поездов. Однако есть более простой способ. Общее время в пути (t) подсчитать несложно — сумма времен остановок на станциях и время перемещений между станциями (скорость — постоянная величина). Время встречи поездов равно $t/2$, если оно не совпадает с моментом остановки. При встрече на какой-то станции требуется сделать небольшое уточнение. Кстати, задача значительно усложняется, если скорость поездов непостоянна (разгон, торможение, остальное время — постоянная скорость). В этом случае необходимо уметь очень аккуратно решать квадратные уравнения. Вернемся к нашей задаче. Рассмотрим, как обычно, пример: V=1.0, M=5.0, N=3 и расстояния 10.0 и 7.0. Общее время в пути 22 единицы. Половина этого значения 11. Но к этому моменту времени второй поезд стоит на станции уже 4 единицы времени, а первый — 1 единицу. Значит, время встречи 11–1 или 10 единиц времени.

14. Имеется N ($10 \le N \le 100$) ламп, пронумерованных от 1 до N. Каждая лампа может иметь два состояния — включена, выключена. Четыре кнопки позволяют управлять лампами следующим образом:

- кнопка 1 — при нажатии все лампы изменяют свое состояние на противоположное;
- кнопка 2 — при нажатии изменяется состояние всех ламп, имеющих нечетные номера;

- кнопка 3 — при нажатии изменяется состояние ламп, имеющих четные номера;
- кнопка 4 — при нажатии изменяется состояние всех ламп, имеющих номера $3k+1 (k \geq 0)$, т. е. 1, 4, 7, ...

Первоначально все лампы включены. Имеется счетчик C ($1 \leq C \leq 10000$), который учитывает суммарное число нажатий всех кнопок.

Требуется по значению числа C и конечному состоянию некоторых ламп определить все возможные, конечные, различные конфигурации N ламп.

Входные данные. Файл *Input.txt* (см. рисунок 6.21) содержит четыре строки:

- количество ламп N;
- значение счетчика C;
- номера тех ламп, которые в конечной конфигурации включены;
- номера тех ламп, которые в конечной конфигурации выключены.

Номера ламп в третьей и четвертой строках записываются через пробел, списки номеров ламп заканчиваются числом −1.

Пример.

Input.Txt

```
10
1
-1
7 -1
```

Output.Txt

```
0000000000
0110110110
0101010101
```

Рис. 6.21

Выходные данные. В файл *Output.txt* (см. рисунок 6.21) записываются все возможные конечные конфигурации (без повторений), каждая — в отдельной строке. Конфигурация — это строка длины N (первый символ — состояние первой лампы, N-й — состояние N-й лампы), где 0 означает, что лампа выключена, а 1 — включена.

Примечание

Количество ламп, о которых известно, что в конечной конфигурации они включены, меньше или равно 2. Гарантируется, что существует хотя бы одна конечная конфигурация для каждого входного файла.

Указание. Заметим, что перебирать все 4^C вариантов, при C=10000 — бесполезное занятие, попытаемся упростить задачу. Во-первых, последовательное действие любых двух кнопок, например, 1–2, эквивалентно их действию в противоположном порядке, т. е. 2–1 (коммутативность). Во-вторых, последовательное применение какой-либо кнопки два раза подряд не изменяет состояние лампочек. Поэтому любое значение C, большее 4, уменьшается на 2. Итак, нечетные C, большие 4, заменяются на C, равное 3, а четные C, большие 5, на C, равное 4. Остается перебрать различные варианты конечных конфигураций при

$C \le 4$, их всего 15 (1; 2; 3; 4; 1,2; 1,3; 1,4; 2,3; 2,4; 3,4; 1,2,3; 1,2,4; 1,3,4; 2,3,4; 1,2,3,4), что, безусловно, не 4^C. Необходимо только не забыть о том, что, например, при C, равном 3, следует проверять и конфигурации, получаемые при C, равном 1.

15. «*Звездная ночь*». Посмотрите на ночное небо, где светящиеся звезды объединены в созвездия (кластеры) различных форм. Созвездие — это непустая группа рядом расположенных звезд, имеющих соседями звезды либо по горизонтали, либо по вертикали, либо по диагонали. Созвездие не может быть частью другого созвездия.

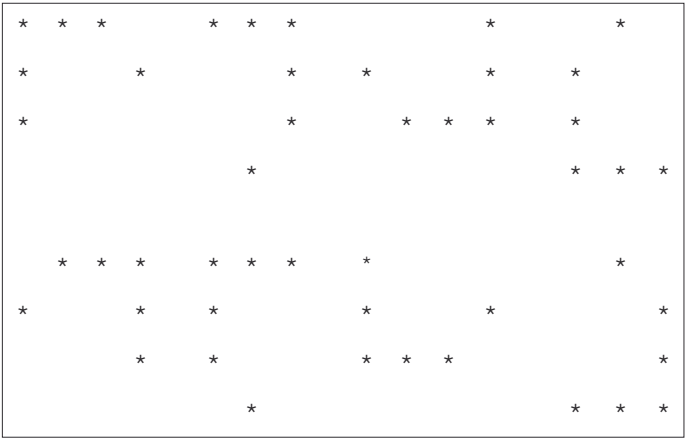


Рис. 6.22

Созвездия на небе могут быть одинаковыми. Два созвездия являются одинаковыми, если они имеют одну и ту же форму независимо от их ориентации. В общем случае число возможных ориентаций для созвездия равно восьми, как показано на рисунке 6.22.

Ночное небо представляется картой неба, которая, в свою очередь, является двумерной матрицей, состоящей из нулей и единиц. Элемент этой матрицы содержит цифру 1, если это звезда, и цифру 0 — в противном случае.

Для заданной карты неба отметить все созвездия, используя при этом маленькие буквы. Одинаковые созвездия должны быть отмечены одной и той же буквой; различные созвездия должны быть отмечены различными маленькими буквами.

Для того чтобы отметить созвездие, необходимо заменить каждую единицу в созвездии соответствующей маленькой буквой.

Входные данные. Входной файл с именем *starry.in* содержит в первых двух строках ширину W и высоту H матрицы, соответствующей карте неба. Далее в этом файле следует H строк, в каждой из которых содержится W символов.

Выходные данные. Файл *starry.out* содержит ту же самую матрицу карты неба, что и файл *starry.in*, только созвездия в ней отмечены так, как описано в условии задачи.

Ограничения:
$0 \le W$(ширина карты неба) ≤ 100,
$0 \le H$(высота карты неба) ≤ 100,
$0 \le$ Число созвездий ≤ 500,
$0 \le$ Число различных созвездий ≤ 26($a..z$),
$1 \le$ Число звезд в каждом созвездии ≤ 160.

Пример
starry.in
23
15

Таблица 6.1

1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1	0	1	1	0	1
0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	1	1	1
0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	0	0	1	0
0	0	0	0	0	1	0	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	1	1	1	0	0	0

starry.out

Таблица 6.2

a	0	0	0	a	0	0	0	0	0	0	0	0	0	b	0	0	0	0	0	0	0
0	a	a	a	a	a	0	0	0	c	c	c	c	c	0	0	0	d	0	d	d	0	d
0	a	0	0	0	0	0	0	0	c	0	0	0	c	0	0	0	d	d	d	d	d	d
0	0	0	0	0	0	0	0	0	c	0	b	0	c	0	0	0	0	0	0	0	0	0
0	0	0	0	0	e	e	e	0	c	0	0	0	c	0	0	0	0	0	0	0	0	0
0	0	0	0	e	0	0	e	0	c	c	c	c	c	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	e	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	b	0	f	0	0	0	0	0	0	c	c	c	c	c	0	0	a	0	0	0	0
0	0	0	0	f	0	0	0	0	0	0	c	0	0	0	c	0	0	a	a	a	a	a
0	0	0	0	0	0	0	d	d	d	0	c	0	b	0	c	0	a	0	0	0	a	0
0	0	0	0	0	b	0	0	d	d	0	c	0	0	0	c	0	0	0	0	0	0	0
0	0	0	g	0	0	0	d	d	d	0	c	c	c	c	c	0	0	0	0	0	0	0
0	0	g	0	0	0	0	d	d	d	0	0	0	0	0	0	0	e	0	0	0	0	0
0	0	0	0	b	0	0	0	d	0	0	0	0	f	0	0	0	e	0	0	e	0	b
0	0	0	0	0	0	0	d	d	d	0	0	0	f	0	0	0	e	e	e	0	0	0

В таблице 6.2 представлен один из возможных вариантов выходного файла для представленного выше входного файла, содержащего карту неба в таблице 6.1.

Указание. Пусть карта описывается массивом *A*:

```
A:Array[0..MaxN+1,0..MaxN+1] Of Byte;
```

где *MaxN*=100, максимальный размер карты. Вместо 1, описывающей звезду созвездия, элементу *A* в процессе обработки будем присваивать код (*ASCII*) соответствующей буквы. Попробуем «набросать» общий алгоритм.

```
...
For i:=1 To N Do
 For j:=1 To M Do
  If A[i,j]=1 Then
Begin
   <Обход созвездия, начиная с клетки (i,j), по принципу обхода в ширину. Результат в переменной Work_Star (описание созвездия) и в переменной d (фиксируется отклонение влево от координаты j)>;
   <Получение всех симметричных созвездий. Проверка совпадения найденного созвездия с ранее полученными. Результат - код ASCII созвездия. Если совпадения не было, то полученное созвездие запоминается>;
    <Пометка полученного созвездия кодом ASCII>;
  End;
...
```

Подумаем об используемой оперативной памяти. Максимальный размер карты — 100×100. Одно созвездие может использовать всю карту, т. е. при описании созвездия необходимо «закладывать» максимальный размер 10000. С другой стороны, созвездий может быть 26, итого требуется 260000 байт. Цифра большая. Необходимо сокращать, использовать побитовое хранение описаний созвездий. Делим на 8 и получаем 32500 байт, что уже допустимо.

Определим структуры данных и сделаем первое уточнение алгоритма решения.

```
Const MaxN=100;{*Максимальный размер карты.*}
        MxCh=26;{*Количество различных созвездий.*}
        StArt=97;{*ASCII код буквы a.*}
        Dx: Array[1..8] Of Integer=( 0, 0, 1, 1,
           1,-1,-1,-1);{*Приращения, используемые
           при обходе в ширину.*}
        Dy: Array[1..8] Of Integer=( 1,-1, 1,
           0,-1, 1, 0,-1);
Type MasSt=Array[0..MaxN+1, 0..(MaxN+8) Div 8] Of
           Byte;{*Карта отдельного созвездия,
           используется побитовое хранение.*}
     Star=Record
          A: MasSt;
          N, M: Integer;
          End;
         {*Описание созвездия.*}
Var A: Array[0..MaxN+1, 0..MaxN+1] Of Byte;
           {*Описание карты.*}
    N, M: Integer;{*Размер карты.*}
    Rs: Array[1..MxCh] Of Star;{*Массив для
           хранения различных созвездий.*}
    Sc: Integer;{*Число различных созвездий.*}
```

При таком определении структур данных основная процедура *Solve* имеет вид.

```
Procedure Solve;
 Var i, j, d: Integer;
     Work_Star: Star;
     c: Byte;
 Begin
  For i:=1 To N Do
   For j:=1 To M Do
    If A[i, j]=1 Then
```

```
    Begin
      d:=GetSt(i,j,Work_Star);{*Обход в ширину.
            d - отклонение влево от начала точки
            обхода. Стандартный обход в ширину
            должен быть модифицирован так, чтобы
            подсчитывался размер созвездия -
            прямоугольник наименьшего размера,
            в который помещается созвездие. Кроме
            того, требуется выполнять битовую
            кодировку созвездия.*}
      c:=Check(Work_Star);{*Проверка на совпадение
            с ранее полученными созвездиями.*}
      SetUp(i,j-d,c,Work_Star);{*Пометка созвездия
            буквой алфавита.*}
    End;
  End;
```

Дальнейшее уточнение алгоритма предоставим читателю. Самая трудоемкая ее часть — проверка созвездий на совпадение.

16. «*Самый длинный префикс*». Структура некоторых биологических объектов представляется последовательностью их составляющих. Эти составляющие обозначаются заглавными буквами. Биологи интересуются разложением длинной последовательности в более короткие последовательности. Эти короткие последовательности называются примитивами. Говорят, что последовательность S может быть образована из данного множества примитивов P, если существует n примитивов $p_1,\ldots,p_n$ в P, таких, что их конкатенация (сцепление) $p_1,\ldots,p_n$ равняется S. При конкатенации примитивы $p_1,\ldots,p_n$ записываются последовательно без разделительных пробелов. Некоторые примитивы могут встречаться в конкатенации более одного раза, и не обязательно все примитивы должны быть использованы.

Например, последовательность *ABABACABAAB* может быть образована из множества примитивов {*A, AB, BA, CA, BBC*}. Первые K символов строки S будем называть префиксом строки S длины K.

Требуется написать программу, которая для заданного множества примитивов P и последовательности T определяет длину максимального префикса последовательности T, который может быть образован из множества примитивов P.

Входные данные (см. рисунок 6.23) расположены в двух файлах. Файл *input.txt* описывает множество примитивов P, а файл *data.txt* содержит последовательность T, требующую проверки. Первая строка *input.txt* содержит n — количество при-

митивов в множестве P ($1 \le n \le 100$). Каждый примитив задан в двух последовательных строках. Первая из них содержит длину L примитива ($1 \le L \le 20$), а вторая — строку из заглавных букв (от '*A*' до '*Z*') длины L. Все n примитивов различны. Каждая строка файла *data.txt* содержит одну заглавную букву в первой позиции. Файл *data.txt* заканчивается строкой с символом '.' (точка) в первой позиции. Длина последовательности не меньше 1 и не больше 500 000.

Выходные данные (см. рисунок 6.23). В первую строку файла *output.txt* требуется записать длину максимального префикса последовательности T, который может быть образован из множества P.

Указание. Введем переменную с именем *MaxPr* типа *LongInt* для хранения значения длины максимального префикса. Ответим на вопрос: как задействовать примитив в сравнениях? Если сравнение начинается с первой позиции последовательности символов, то понятно — примитивы начинаются с этой же буквы. А если нет?

Мы в какой-то позиции последовательности с номером t. Новое подмножество примитивов для сравнения выбирается только в том случае, если *MaxPr* равно $t-1$!!! Это означает, что закончено сравнение по какому-то примитиву, он полностью входит в конкатенацию, и есть полное право подключать к процессу новое подмножество примитивов. Если $t-1$ не равно *MaxPr*, то процесс сравнения символов последовательности продолжается только с символами ранее выделенных примитивов. В том случае, когда множество выделенных примитивов пусто, обработка закончена. То, что мы нашли, — значение *MaxPr* — ответ задачи.

Пример.

input.txt

```
5
1
A
2
AB
3
BBC
2
CA
2
BA
```

data.txt

```
A
B
A
B
A
C
A
B
A
A
B
C
B
.
```

output.txt

```
11
```

Рис. 6.23

А в какие моменты времени изменяется значение *MaxPr*? Как только заканчивается сравнение хотя бы с одним примитивом, изменяем значение *MaxPr*.

Для хранения «активных» примитивов следует ввести структуру данных P — массив из M строк и *MaxN* плюс один столбец (*Array*[1..*M*,0..*MaxN*] *of Integer*). Почему M? Нет необходимости хранить предысторию процесса сравнения на большую глубину, ибо максимальная длина примитива M. В элементе же нулевого столбца P фиксируем количество записей в соответствующую строку. С этой структурой данных следует предусмотреть следующие операции: вставка элемента (открытие нового примитива,

перевод его в состояние «активный»); исключение элемента (сравнение очередного символа активного примитива и строки дало отрицательный результат); сжатие данных (если действительно для хранения активных примитивов выбран массив).

17. Дан текст, набранный символами одинаковой ширины (включая пробелы). Длина строки при печати этим шрифтом на листе бумаги равна S. Назовем *пустой* последовательность пробелов между соседними словами в строке, а также от начала строки до первого слова в ней и от последнего слова в строке до конца строки.

Требуется преобразовать текст так, чтобы сумма кубов *пустот* по всем строкам была минимальна.

Для достижения этой цели разрешается заменять произвольную *пробельную последовательность* (непустую последовательность подряд идущих пробелов и/или символов перевода строки) любой другой последовательностью.

Входные данные. Первая строка входного файла содержит целое число S ($1 \le S \le 80$). В последующих строках записан текст, содержащий не более 500 слов. Длина каждого слова не превосходит S.

Выходные данные. В первую строку выходного файла записывается минимально возможная сумма кубов пустот по всем строкам. В последующих строках выводится преобразованный текст.

input.txt	*output.txt*
10 aaa-bbb---bbbb aaaa ccc---------cccc	15 -aaa--bbb-bbb b-aaaa-ccc-ccc c-
10 aa-bb---bb aa cc---------cc	8 -aa-bb-bb-aa-c c-cc-
10 aaaa-bbbb---bbbb aaaa cccc---------cccc	6 aaaa-bbbb-bbb b-aaaa-cccc-cc cc-
13 aaaaa-aaaa-aaa-aa-a	35 -aaaaa—aaaa-- aaa—aa--a--
13 aaaaaa-aaaaa-aaaa-aaa-aa-a	5 aaaaaa-aaaaa- aaaa-aaa-aa-a
13 aaaaaaa-aaaaaa-aaaaa-aaaa- aaa-aa-a	59 ---aaaaaaa---a aaaaa-aaaaa--a aaa-aaa-aa-a-

Рис. 6.24

Для осознания сути задачи на рисунке 6.24 приведены примеры входных и соответствующих выходных файлов (в таблице символом «-» обозначен пробел — удобнее считать).

Указание. Решение основано на факте — сумма кубов минимальна в том случае, когда количество пробелов между словами совпадает, и если этого достичь нельзя, то эти значения (количество пробелов) наиболее близки друг другу. Пусть мы размещаем в строке q слов, количество символов в словах равно w. Длина строки известна — S позиций. Обязательное количество пробелов на каждом месте (d), а их (q+1), подсчитываем, находя целую часть частного от деления количества пробелов (S–w) на количество мест q+1. Оставшееся число пробелов (m:=(S–w) *Mod* (q+1)) необходимо распределить по одному к пробелам на m местах.

18. «*Парковка*»*. Парковка около Великой Стены состоит из длинного ряда парковочных мест, полностью заполненного машинами. Один из краев считается левым, а другой — правым. Каждая из машин имеет свой тип. Типы перенумерованы натуральными числами, и несколько машин могут иметь один и тот же тип. Рабочие решили упорядочить машины в парковочном ряду так, чтобы их типы возрастали слева направо. Упорядочение состоит из нескольких этапов. В начале каждого этапа рабочие одновременно выводят машины с их мест и затем ставят машины на свободные места, образовавшиеся в результате выезда машин на данном этапе, т. е. меняют их местами. Каждый рабочий перемещает только одну машину за этап, и некоторые рабочие могут не участвовать в перемещении машин на этапе. Эффективность процесса упорядочивания машин определяется количеством этапов.

Предположим, что на парковке находится N машин и W рабочих.

Требуется написать программу, которая находит такое упорядочивание, при котором количество этапов не превосходит значения $N/(W-1)$, округленного в большую сторону, т. е. $\lceil N/(W-1) \rceil$. Известно, что минимальное число этапов не превосходит $\lceil N/(W-1) \rceil$.

Рассмотрим следующий пример. На парковке находятся 10 машин типов 1, 2, 3, 4 и четверо рабочих. Начальное положение машин слева направо, заданное их типами — 2 3 3 4 4 2 1 1 3 1.

---

* Задача с Международной олимпиады школьников 2000 года.

Минимальное количество этапов — 3, и они могут быть выполнены так, что расположение машин по типам после каждого из этапов следующее:

- 2 1 1 4 4 2 3 3 3 1 — после первого этапа,
- 2 1 1 2 4 3 3 3 4 1 — после второго этапа,
- 1 1 1 2 2 3 3 3 4 4 — после третьего этапа.

Входной файл имеет имя *car.in* (см. рисунок 6.25). Первая строка входного файла состоит их трех чисел. Первое — количество машин N, $2 \le N \le 20000$. Второе — количество типов машин M, $2 \le M \le 50$, причем тип машины является натуральным числом от 1 до M. На парковке есть хотя бы одна машина каждого типа. Третье — количество рабочих W, $2 \le W \le M$. Вторая строка содержит N целых чисел, где i-е число является типом i-й машины в парковочном ряду, считая слева направо.

Пример.

car.in

```
10 4 4
2 3 3 4 4 2 1 1 3 1
```

car.out

```
3
4 2 7 3 8 7 2 8 3
3 4 9 9 6 6 4
3 1 5 5 10 10 1
```

Рис. 6.25

Выходной файл имеет имя *car.out* (см. рисунок 6.25) Первая строка выходного файла содержит число R — количество этапов. Далее следует R строк, где i+1 строка выходного файла описывает i-й этап. Формат строки, описывающей этап, следующий. Первое число — количество машин C, перемещаемых на данном этапе. Далее следует C пар чисел, определяющих перемещения машин на данном этапе. Первое число пары — позиция, с которой машина перемещается, второе число пары — позиция, на которую машина перемещается. Позиция машины является целым числом от 1 до N и отсчитывается с левого конца ряда.

В случае, если существует несколько оптимальных решений, выдайте только одно из них.

Оценка решения. Предположим, что количество этапов, выданных вашей программой, — R, а $\lceil N/(W-1) \rceil = Q$. Если какой-то из R этапов выдан некорректно или после выполнения всех этапов типы машин не выстраиваются в возрастающем порядке, вы набираете 0 очков за тест. В противном случае ваши очки будут начислены в соответствии со следующими правилами:

- $R \le Q$ 100% очков
- $R = Q+1$ 50% очков
- $R = Q+2$ 20% очков
- $R >= Q+3$ 0% очков

Указание. Прагматика и еще раз прагматика торжествует в решении данной задачи. Поиск минимального количества этапов в задаче не требуется. Необходимо осуществить перестановку машин на стоянке за Q этапов — это дает 100%-ный балл. Если за один этап на свои места ставится $W-1$ машина, то этот результат достигается. Так как на стоянке работают W рабочих, $W-1$ машина всегда ставится на свои места на одном этапе, с помощью простого эвристического алгоритма.

- Взять машину, стоящую не на своем месте, и переставить ее на место в том промежутке, где она должна стоять, занятое машиной, также стоящей не на своем месте;
- продолжить этот процесс (с последней машиной) до тех пор, пока не окажутся задействованными в перестановке все рабочие или не получится цикл. При этом машина, вытесненная последней, ставится на первоначальное место, с которого начинался процесс перестановки. Последняя машина не обязательно попадает в свой промежуток. В случае цикла и оставшегося количества рабочих более одного, алгоритм повторяет свою работу с этими рабочими над машинами, стоящими, естественно, не на своих местах.

Для примера из формулировки задачи этот алгоритм дает этапы работы, приведенные в таблице 6.3.

Таблица 6.3

№ этапа	Место									
	1	2	3	4	5	6	7	8	9	10
1	**2**	3	3	**4**	4	**2**	1	1	**3**	1
2	**2**	3	3	2	**4**	3	1	1	4	**1**
3	1	**3**	**3**	2	2	3	**1**	**1**	4	4
	1	1	1	2	2	3	3	3	4	4

Ответ (файл *car.out*):

```
4
4 1 4 4 9 9 6 6 1
4 1 5 5 10 10 1 2 2
4 2 7 7 2 3 8 8 3
1 1 1
```

Решение состоит не из минимального числа этапов, но оно отвечает требованию задачи.

19. «*Медианная энергия*»*. В новом космическом эксперименте участвуют N объектов, пронумерованных от 1 до N. Изве-

* Задача с Международной олимпиады школьников 2000 года.

стно, что N — нечетно. Каждый объект имеет уникальную, но неизвестную энергию Y, выражающуюся натуральным числом, $1 \le Y \le N$. Назовем объектом с медианной энергией такой объект X, для которого количество объектов с энергией, меньшей, чем у X, равно количеству объектов с энергией, большей, чем у X.

Требуется написать программу, которая определяет объект с медианной энергией. К сожалению, сравнивать энергии можно только с помощью устройства, способного для трех различных заданных объектов определить, какой из них имеет медианную энергию.

Библиотека. Вам предоставляется библиотека *device* с тремя операциями:

- *GetN*, вызывается только один раз в начале программы; не имеет аргументов; возвращает значение N;
- *Med3*, вызывается с тремя различными номерами объектов в качестве аргументов; возвращает номер объекта, имеющего медианную (среднюю) энергию;
- *Answer*, вызывается только один раз в конце программы; единственный аргумент — номер объекта X, имеющего медианную энергию. Эта функция корректно завершает выполнение вашей программы.

Библиотека *device* генерирует два текстовых файла *median.out* и *median.log*. Первая строка файла *median.out* содержит одно целое число: номер объекта, переданного библиотеке при вызове *Answer*. Вторая строка содержит одно целое число: количество вызовов *Med3*, которое было сделано вашей программой. Диалог между вашей программой и библиотекой записывается в файл *median.log*.

Инструкция для программирующих на языке Pascal: включите строку *uses device*; в текст вашей программы.

Экспериментирование. Вы можете поэкспериментировать с библиотекой, создав текстовый файл *device.in*. Файл обязан содержать две строки. Первая строка — одно целое число: количество объектов N. Вторая строка — целые числа от 1 до N в некотором порядке: i-е число является энергией объекта с номером i.

Пример
device.in
5
2 5 4 3 1

Файл *device.in*, указанный выше, описывает 5 объектов с их энергиями. Эту информацию можно представить в виде таблицы 6.4.

Таблица 6.4

Номер объекта	1	2	3	4	5
Энергия	2	5	4	3	1

Ниже приведена корректная последовательность с пятью обращениями к библиотеке:

1. *GetN* возвращает 5.
2. *Med3*(1,2,3) возвращает 3.
3. *Med3*(3,4,1) возвращает 4.
4. *Med3*(4,2,5) возвращает 4.
5. *Answer*(4)

Ограничения:

- количество объектов N нечетно, и $5 \le N \le 1499$;
- для объекта с номером i выполняется $1 \le i \le N$;
- для энергии объекта Y выполняется $1 \le Y \le N$, и все энергии различны;
- библиотека для *Pascal* содержится в файле: *device.tpu*
- объявление функций и процедуры на *Pascal*:
  Function GetN:Integer;
  Function Med3(*x,y,z:Integer*):*Integer;*
  Procedure Answer(*m:Integer*);
- в ходе исполнения программы разрешается использовать не более 7777 вызовов функции *Med3*;
- ваша программа не должна читать или писать какие-либо файлы.

Указание. Пусть есть массив с энергиями объектов (см. таблицу 6.5).

Таблица 6.5

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	5	4	13	10	2	11	12	9	6

Все, что мы можем сделать с помощью операции *Med3*, это разбить массив на три части, выбрав случайным образом два несовпадающих номера (a и b) элементов. Не деление пополам, как в бинарном поиске, а деление на три части — «тринарный» поиск. Пусть a=5 и b=12. Просматриваем все номера объектов, помеченные признаком *True*. В начальный момент времени для всех объектов этот признак равен *True*. Если операция *Med3(a,b,i)* дает в качестве результата:

- значение a, то помечаем объект с номером i как элемент, принадлежащий левой части (признак *_Left*);

- значение *i*, то помечаем объект с номером *i* как элемент, принадлежащий средней части (признак *_Middle*);
- значение *b*, то помечаем объект с номером *i* как элемент, принадлежащий правой части (признак *_Right*).

В таблице 6.6 буквами *L*, *M*, *R* обозначены признаки *_Left*, *_Midlle*, *_Right*. Буквой *N* обозначен признак *_None*, элемент не рассматривается на данном шаге анализа.

Таблица 6.6

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Признак	*M*	*L*	*L*	*M*	*N*	*L*	*R*	*R*	*L*	*R*	*R*	*N*	*M*

Итак, разбили, а что дальше? Напрашивается единственно разумный ход — выбрать одну из трех частей и продолжить поиск (другого не дано).

Вся сложность задачи — в выборе параметров поиска и в аккуратном переходе от одного этапа поиска к другому. Какую часть выбрать для следующего этапа поиска? Мы имеем в левой части результата поиска 4 элемента, в правой — 4 и в средней (без учета *a* и *b*) — 3. Первоначально осуществлялся поиск элемента с медианной энергией 7 в массиве из 13 элементов. После этого этапа необходимо искать элемент со вторым значением энергии среди элементов, помеченных признаком *_Middle*. Часть параметров поиска определена: количество и признак элементов, среди которых ищется искомый; номер значения энергии объекта. Еще один параметр — одно из предыдущих значений *a* или *b* — требует пояснений (это самый сложный момент). Поясним рисунком 6.26.

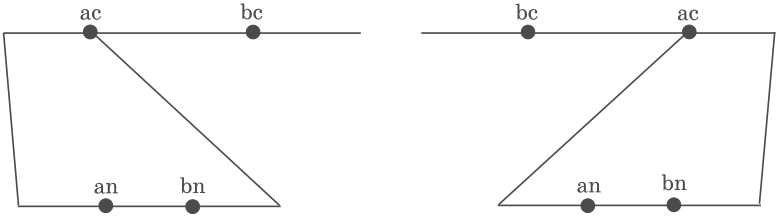


Рис. 6.26

Оказалось, что поиск следует продолжать в левой (по отношению к *a*) части. Генерируем новые значения *a b* (обозначены как *an, bn*). Если после этого средний элемент (*Med3(an,bn,ac)*)

находится не на месте *bn*, то структура последующего поиска нарушается. Необходимо поменять значения у *an* и *bn*. Так, если после первого этапа поиска мы выяснили, что его следует продолжать в левой части, то и следующее деление на три части должно осуществляться с учетом этого результата.

Примечания.

1. Прервите чтение и проделайте ручной просчет поиска по данным из условия задачи без смены значений. Результат уже на этом простом примере окажется отрицательным.

2. Сделайте еще два рисунка и выясните особенности использования функции *Med3* при продолжении поиска в средней и правой частях.

Позволим себе привести полный текст решения этой красивой задачи*.

```
Program Median;
  Uses device;
  Const   MaxN = 1500;
  Type  TDir = (_None, _Middle, _Left, _Right);
  Var  n: Integer;
    Mas : Array [1..MaxN] Of TDir;
    Lt, Rt : Integer;  (*Количество элементов
       в подмножествах *)
  Procedure MakeMiddle(Dir: TDir);{*Помечаем
     элементы, среди которых осуществляется поиск.*}
    Var i: Integer;
    Begin
      For i := 1 To N Do
       If Mas[i] = Dir Then Mas[i]:= _Middle
                       Else Mas[i]:= _None;
    End;
  Procedure Swap(Var a,b: Integer);
    Var c: Integer;
    Begin  c := a; a := b; b := c; End;
  Procedure GetNext(Var a,b: Integer; len: Integer);
        {*Генерируем номера a и b.*}
    Var i,t,ca,cb: Integer;
    Begin
      t := 0;
```

* Это одна из версий программы решения задачи. Разработана Дмитрием Сергеевичем Шулятниковым.

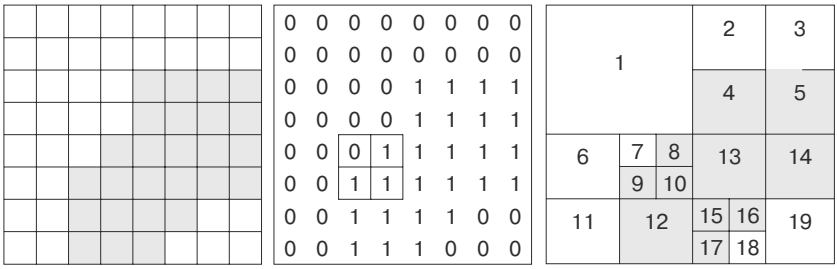
```
    ca := Random(len)+1;
    cb := Random(len)+1;
    While ca=cb Do cb := Random(len)+1;
    For i := 1 To N Do {*Сложность в том, что номера
          оставшихся элементов для поиска идут
          не подряд.*}
      If Mas[i] = _Middle Then
        Begin
          inc(t);
          If ca = t Then a := I;
          If cb = t Then b := i;
        End;
 End;
Procedure Doit(Dir: TDir; res, len, lb: Integer);
        {*Основная процедура, параметры поиска
        описаны ранее.*}
 Var a, b, i, t: Integer;
 Begin
  MakeMiddle(Dir); {*Выделяем элементы, среди
        которых осуществляется поиск.*}
  If len = 1 Then
    Begin
      a := 1;
      While Mas[a] <> _Middle Do inc(a);
      Answer(a);
    End
            Else
    Begin
      GetNext(a,b,len);
      If (Dir=_Right)
        Then
          Begin
            If Med3(lb,a,b) <> a
              Then Swap(a,b)
          End
        Else
          If Med3(a,b,lb) <> b
            Then Swap(a,b);
            {*Корректируем значения a и b с учетом
            предыдущего поиска. Пояснение приведено
            на рисунке в тексте.*}
      If len = 2 Then
```

```
    Begin
      If res = 1 Then Answer(a)Else Answer(b)
    End
              Else
    Begin
       lt := 0; rt := 0;
       Mas[a] := _None; Mas[b] := _None;
       For i := 1 To n Do
        If Mas[i] = _Middle Then
          Begin
            t := Med3(a,b,i);
            If t = b
             Then
               Begin inc(rt); Mas[i] := _Right End
               {*Пишем в правую часть.*}
             Else
               If t = a Then
                Begin inc(lt); Mas[i]:= _Left End;
          End;
       If Res = lt+1
         Then Answer(a)
         Else {*Искомый
            элемент попал на границу интервала,
            поиск завершен.*}
          If Res = len-rt
            Then Answer(b)
            Else
             If res < lt+1 Then Doit(_Left,res,lt,a)
            {*Поиск с новыми параметрами.*}
                          Else
            If res < len-rt
             Then Doit(_Middle, res-lt-1,
                       len-lt-rt-2,b)
             Else Doit(_Right,res-len+Rt,Rt,b);
    End;
  End;
End;
Procedure Solve;
  Begin
    n := GetN; Randomize;
    FillChar(Mas, SizeOf(Mas), _Middle);
    Doit(_Middle,n shr 1+1, n, 1);{*Первый вызов, n
          shr 1+1 - среднее значение энергии, 1-
```

```
                условно считаем, что ранее был первый
               номер (обеспечивает правильный поиск).*}
  End;
Begin Solve End.
```

20. В компьютерной графике* для обработки черно-белых изображений используется их преобразование в строку из десятичных чисел. Изображение или его часть (квадрат меньшего размера) последовательно делится на 4 равных квадрата. Если все точки в квадрате одного цвета (черного или белого), то процесс деления этой части изображения заканчивается. Квадраты, содержащие как черные, так и белые точки, опять делятся на 4 равных квадрата. Процесс продолжается до тех пор, пока каждый из квадратов не будет содержать точки только одного цвета.

Например, если использовать 0 для белого и 1 для черного, то изображение на рисунке 6.27(а) будет записано матрицей из нулей и единиц, как представлено на рисунке 6.27(б).



(а) *(б)* *(в)*

Рис. 6.27

Матрица будет разделена на квадраты, как показано на рисунке 6.27(в), где квадраты, выделенные серым, те, которые содержат только черные точки.

Преобразование выполняется следующим образом. Корень дерева представляет все изображение. Каждая вершина дерева описывает некий квадрат. Она может иметь 4 исходящих ребра к другим вершинам, представляющим квадраты, на которые делится исходный квадрат. Вершины без исходящих ребер со-

---

* Одна из самых красивых задач, с которой пришлось работать автору.

ответствуют квадратам, состоящим из точек одного цвета. Например, приведенному выше изображению на рисунке 6.27(в) с разбивкой на квадраты соответствует дерево, изображенное на рисунке 6.28. Серые вершины обозначают квадраты, содержащие 2 цвета, а белые и черные — квадраты, состоящие из точек одного цвета (такого же, как цвет квадрата).

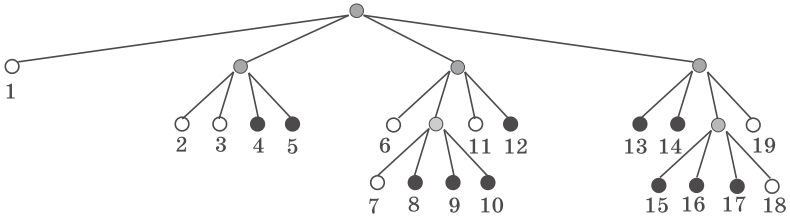


Рис. 6.28

В дереве вершины пронумерованы в соответствии с нумерацией квадратов на рисунке 6.27(в). Квадраты обходятся слева-направо и сверху-вниз, начиная с верхнего левого угла (верхний левый (В–Л), верхний правый (В–П), нижний левый (Н–Л), нижний правый (Н–П)). Каждая черная вершина дерева кодируется следующим образом: поднимаемся от вершины по ребрам к корню, записывая при этом подряд цифры от 1 до 4 в соответствии с тем, где находится текущий квадрат по отношению к вышестоящему (верхний левый — 1, верхний правый — 2 , нижний левый — 3, нижний правый — 4). Получившееся число записано по основанию 5, запишем его по основанию 10.

Например, вершина с номером 4 имеет путь Н-Л, В-П. Получается число 32_5 (по основанию 5) или 17_{10} (по основанию 10). Вершина с номером 12 имеет путь Н-Л, Н-П, или 43_5 и 23_{10} соответственно. Вершина с номером 15 имеет путь В-Л, Н-Л, Н-П, или $134_5=44_{10}$. После этого дерево кодируется строкой чисел: 9 14 17 22 23 44 63 69 88 94 113.

Требуется написать программу перевода изображения в строку чисел и обратного преобразования — строки чисел в изображение.

Входные данные. Файл содержит описание одного или нескольких изображений. Все изображения — это квадратные рисунки, сторона квадрата которых имеет значение длины, являющееся степенью двойки. Входной файл начинается с целого числа

n, где $|n|$ — длина стороны квадрата ($|n| < 64$). Если число n больше 0, то затем следует $|n|$ строк по $|n|$ знаков в строке, заполненных 0 и 1. При этом 1 соответствует черному цвету. Если n меньше 0, то затем следует описание изображения в виде строки из десятичных чисел, оканчивающейся −1. Полностью черному квадрату соответствует кодировка из одного 0. Белый квадрат имеет пустую кодировку (ничего не вводится). Признаком конца входного файла является значение n, равное 0.

Выходные данные. Для каждого изображения из входного файла выводится его номер. В том случае, когда изображение задается с помощью 0 и 1, в выходной файл записывается его представление в виде строки из десятичных чисел. Числа в строке сортируются в порядке возрастания. Для изображений, содержащих больше 12 черных областей, после каждых 12 чисел вывод начинается с новой строки. Количество черных областей выводится после строки из десятичных чисел. В том случае, когда изображение задается строкой из десятичных чисел, в выходной файл записывается его представление в виде квадрата, в котором символ '.' соответствует 0, а символ '*' — 1. Пример входного и выходного файлов приведен в таблице 6.7.

Таблица 6.7

Входной файл	Выходной файл
8	Изображение 1
0 0 0 0 0 0 0 0	9 14 17 22 23 44 63 69 88 94 113
0 0 0 0 0 0 0 0	Общее число черных областей 11
0 0 0 0 1 1 1 1	Изображение 2
0 0 0 0 1 1 1 1	. . . . .
0 0 0 1 1 1 1 1	. . . . .
0 0 1 1 1 1 1 1	. . . . * * * *
0 0 1 1 1 1 0 0	. . . . * * * *
0 0 1 1 1 0 0 0	. . . * * * * *
-8	. . * * * * * *
9 14 17 22 23 44 63 69 88 94 113 -1	. . * * * * . .
2	. . * * * . . .
0 0	Изображение 3
0 0	Общее число черных областей 0
-4	Изображение 4
0 -1	* * * *
	* * * *
	* * * *
	* * * *

Указание. Очередной раз используем технологию «сверху вниз» в наших рассуждениях.

Основная программа не требует пояснений.

```
Begin
 …
 Readln(N);{*Вводим размерность изображения.*}
 While N<>0 Do
   Begin
    ...
    If N>0 Then SolveA Else SolveB;{*Первая
    процедура преобразует изображение в виде матрицы
    в строку десятичных чисел, вторая процедура
    выполняет обратное преобразование.*}
   End;
 ...
End.
```

Дальнейшее уточнение требует осознания того, каким образом в нашей задаче описывается очередной квадрат. Итак, это координаты левого верхнего угла квадрата, длина его стороны и тот путь по изображению, который предшествовал попаданию в этот квадрат.

Формирование пути по изображению следует из формулировки задачи — строка из цифр от 1 до 4.

Назовем функцию обработки квадрата *RecA*, ее заголовок —

```
Function RecA(i,j,d:Integer;Var Way:String):Boolean;
```

Значение функции *True* говорит о том, что квадрат черный.

Другим очевидным соображением является введение массива для хранения результата — десятичных чисел, описывающих изображение. Пусть это будет массив

```
Cp:Array[1..MaxCn] Of LongInt;
```

где *MaxCn* — константа, равная 64×64, а счетчиком числа записей в *Cp* является значение переменной *Cnt*.

Мы совсем забыли сказать о том, что исходное изображение записано в массиве

```
A:Array[1..MaxN,1..MaxN] Of Boolean;
```

где *MaxN* — константа, равная 64. Значение элемента массива *A*[*i*,*j*], равное *True*, соответствует черной клетке с координатами (*i*,*j*), а при *False* — белой. Итак, *SolveA*.

```
Procedure SolveA;
 Var i:Integer;
 Begin
  ReadA;{*Вводим описание изображения из входного
        файла, формируем A.*}
  Cnt:=0;
  If RecA(1,1,N,'') Then
    Begin Cnt:=1; Cp[Cnt]:=0; End;
          {*Если все изображение состоит из
          одного черного цвета, то записываем в
          первый элемент массива Cp ноль
          и заканчиваем обработку этого теста, во
          всех остальных случаях массив Cp
          сформирован рекурсивной функцией RecA.*}
  Sort;{*Сортируем массив Cp.*}
  PrintA;{*Выводим результат в выходной файл
          OutPut.*}
 End;
```

Функция *RecA* имеет вид.

```
Function RecA(i,j,d:Integer;Way:String):Boolean;
 Var k:Integer; c:Boolean;
 Begin
  If d=1
    Then c:=A[i,j] {*Дошли до квадрата
          единичной длины, значение функции
          определяется цветом квадрата.*}
    Else
      Begin
        k:= d Div 2;
        c:=RecA(i,j,k,'1'+Way) And RecA(i,j+k,k,'2'+Way)
         And RecA(i+k,j,d,'3'+Way)
          And RecA(i+k,j+k,d,'4'+Way);
           {*Значение функции на данном уровне
            рекурсии определяется значениями функции
            на квадратах меньшего размера.*}
        If c Then Dec(Cnt,4);{*Если все составляющие
           квадраты черные, то и данный квадрат
           черный.*}
      End;
  If c Then Begin
   Inc(Cnt);{*Квадрат черный, преобразуем путь
          (строка, описывающая число в пятеричной
          системе счисления) в десятичное число и
          записываем результат.*}
```

```
    Cp[Cnt]:=<преобразовать строку символов,
описывающую число в пятеричной системе счисления, в
число в десятичной системе счисления>;
  End;
  RecA:=c;
End;
```

На этой ветке алгоритма осталось уточнить функцию преобразования пути по квадрату в десятичное число.

Процедуры *Sort* и *PrintA* (мы вызываем их из процедуры *SolveA*) «прозрачны» для человека, прочитавшего книгу до этого места. Описание алгоритма получения матрицы, описывающей изображение, из строки десятичных чисел чуть сложнее. Вся суть в аналогичной процедуре *RecB*. Квадрат описывается координатами верхнего левого угла и длиной стороны, безусловно, что они являются параметрами процедуры. Кроме того, у нас есть путь, представленный пятеричным числом. Вопрос. Как из очередной цифры этого числа сделать переадресацию по квадрату, т. е. перейти к одному из четырех квадратов меньшего размера? Ответив на него, можно программировать как процедуру *RecB*, так и весь алгоритм *SolveB*. Поясним ситуацию рисунком 6.29. Пусть координаты левого верхнего угла квадрата задаются значениями переменных *i* и *j*, а длина стороны нового квадрата — значение *r*. Цифры в кружках — это значения переменной *k*, получаемой из очередной цифры пятеричного числа с помощью операции

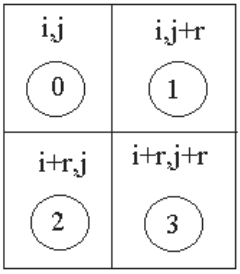


Рис. 6.29

```
k:=Way Mod 5-1;
```

Итак, в зависимости от значения переменной *k* с помощью операций

```
i+r*(k Div 2)   и   j+r*(k Mod 2)
```

мы переходим к левым верхним углам любого из четырех квадратов меньшего размера. Для убедительности проиллюстрируем этот фрагмент рассуждений примером из таблицы 6.8.

Таблица 6.8

Номер вызова процедуры *RecB*	*i*	*j*	*d*	*Way*	*k*	*r*
1	1	1	8	32_5 или 17_{10}	1	4
2	1	5	4	3	2	2
3	3	5	2	0	-	-

Дальнейшую работу по уточнению алгоритма оставим заинтересованному читателю.

21. На числовой прямой отметили несколько отрезков, каждый из которых обозначили одной из букв латинского алфавита (различные отрезки — различными буквами). Используя эти значения как операнды, а также операции объединения (+), пересечения (*), разности (−) множеств и круглые скобки, было составлено теоретико-множественное выражение.

Множество точек, являющееся результатом вычисления составленного выражения, разбивается на некоторое количество связных частей, каждая из которых является либо отдельной точкой, либо промежутком, т. е. отрезком, интервалом или полуинтервалом. Если потребовать, чтобы число частей в разбиении было минимальным (никакие две части нельзя объединить в большую), то такое представление окажется единственным.

Написать программу, представляющую результирующее множество точек в указанном виде.

Входные данные. В первой строке входного файла (*input.txt*, см. рисунок 6.30) содержится натуральное число N — количество отрезков на прямой. В каждой из следующих N строк находится описание одного из отрезков: буква, приписанная этому отрезку, и координаты его левого и правого концов. Все координаты — целые числа из диапазона $[0, 10^9]$, строчные и прописные буквы считаются различными. В следующей строке содержится выражение длиной не более 250 символов. Выражение не содержит пробелов и лишних скобок.

Выходные данные. В выходной файл (*output.txt*, см. рисунок 6.30) записывается разбиение результирующего множества на связные части. Порядок вывода частей должен соответствовать порядку их следования на числовой прямой, если передвигаться по ней слева направо. Точка представляется указанием ее целочисленной координаты в фигурных скобках, числовые промежутки представляются согласно стандартным правилам: квадратная скобка означает, что соответствующая концевая точка принадлежит промежутку; круглая — не принадлежит. Целочисленные координаты левого и правого концов промежутка должны быть разделены точкой с запятой, каждую из частей разбиения следует записать в отдельную строку выходного файла.

Пример.

input.txt

```
3
A 0 30
a 10 15
b 15 20
(A−(a+b))+(a*b)
```

output.txt

```
[0;10)
{15}
(20;30]
```

Рис. 6.30

Указание. Заданные отрезки определяют множество точек на прямой, назовем их граничными. Так, для примера на рисунке 6.29 это [0, 10, 15, 15, 20, 30] (после сортировки).

В результате выполнения теоретико-множественного выражения интервал между двумя соседними граничными точками, например (0, 10), или принадлежит, или не принадлежит результирующему множеству (не может часть интервала (0, 10) принадлежать результату, а другая нет). Для проверки принадлежности интервала достаточно проверить, а принадлежит ли его средняя точка результату. Множество граничных точек преобразуется и имеет вид: [0, 5, 10, 12.5, 15, 15, 15, 17.5, 20, 25, 30]. Количество точек для N интервалов — $N\times4-1$.

Что дальше? Берем каждую точку и проверяем, анализируя выражение. Операция объединения ($A+B$) заменяется на (*A Or B*), ($A-B$) — на (*A And Not B*) и ($A*B$) — на (*A And B*). Запись выражения в форме Бэкуса-Наура имеет вид:

```
<выражение>=<слагаемое>{<операция><слагаемое>}
<операция>=+|-|*
<слагаемое>=(<выражение>)|<идентификатор>
<идентификатор>=a|b|c..z|A|B|C..Z
```

Определим «узловые» моменты решения задачи.

Из структуры данных (они даны не все) приведем следующие.

```
Const MaxN=52;{*Максимальное количество интервалов,
      26 символов алфавита.*}
Var   OnX:Array[1..MaxN*2] Of LongInt;{*Начальное
      множество граничных точек.*}
      N:Integer;{*Количество интервалов.*}
      Numb:Integer;{*Рабочая переменная, определяет
      номер символа в выражении при его анализе.*}
```

Мы должны просматривать все множество граничных точек, их $N\times4-1$, а в массиве *OnX* только часть из них, средние точки интервалов не присутствуют. Не будем дописывать их в *OnX*, а попытаемся просто вычислять по номеру с помощью следующей функции.

```
Function Get(k:Integer):Real;
 Begin
  If k Mod 2=0
    Then Get:=(OnX[k Div 2]*1.0+OnX [k Div 2+1])/2
```

```
  Else Get:=OnX[k Div 2+1];
End;
```

Результат вызова функции *Get* на множестве граничных точек из нашего примера представлен во 2-м столбце таблицы 6.9.

Таблица 6.9

k	*Get(k)*	*AtIn(Get(k))*
1	0	*True*
2	(0*1.0+10)/2=5	*True*
3	10	*False*
4	(10*1.0+15)/2=12.5	*False*
5	15	*True*
6	(15*1.0+15)/2=15	*True*
7	15	*True*
8	(15*1.0+20)/2=17.5	*False*
9	20	*False*
10	(20*1.0+30)/2=25	*True*
11	30	*True*

Функция *AtIn* дает истину или ложь в зависимости от того, принадлежит или нет точка прямой результирующему множеству. Ее вид:

```
Function AtIn(x:Real):Boolean;
 Begin
  Numb:=0;{*Номер обрабатываемого символа
          в выражении.*}
  AtIn:=Expr(x);{*Перенесем очередной раз тяжесть
        на следующую функцию (не будем пока ее
        рассматривать) - вычисления выражения, это
        позволит нам до конца выяснить отношения с
        процедурой WriteRes.*}
 End;
```

Итак, основной алгоритм обработки заключен в процедуре *WriteRes*.

```
Procedure WriteRes;
 Var  a,b:Integer;
      i:Integer;
 Begin
  i:=1;
```

```
  While (i<=N*4-1) And (Not AtIn(Get(i))) Do
    Inc(i); {*Пропускаем граничные точки до тех пор,
         пока не дойдем до точки, принадлежащей
         результирующему множеству.*}
  While i<=N*4-1 Do
    Begin
     a:=i;{*Левая граница интервала из результирующего
          множества.*}
     While (i<=N*4-1) And AtIn(Get(i)) Do Inc(i);
         {*Пропускаем точки, принадлежащие
          результирующему множеству.*}
     b:=i;{*Правая граница интервала.*}
     SayRes(a,b);{*Выводим интервал. Процедура
         не приводится.*}
     Inc(i);
     While (i<=N*4-1) And (Not AtIn(Get(i))) Do
      Inc(i); {*Снова пропускаем граничные точки до
      тех пор, пока не дойдем до точки, принадлежащей
      результирующему множеству.*}
    End;
End;
```

Закончим рассмотрение. Осталось уточнить процедуру *SayRes* — вывода найденного интервала и вычисления выражения *Expr* при конкретном значении x, а также основной алгоритм и ввод данных.

22. Дана последовательность, состоящая из нулей и единиц. Требуется найти N ($0<N\le 20$) битовых подпоследовательностей длиной от A до B ($0<A\le B\le 12$) включительно, которые встречаются в исходной последовательности наиболее часто.

Входные данные. Файл *input.txt* (см. рисунок 6.31) содержит данные в следующем формате. В первой строке — число A, во второй — B, в третьей — число N, в четвертой — последовательность символов 0 и 1, заканчивающуюся символом 2.

Примечание

Размер входного файла не превышает 2-х мегабайт.

Выходные данные. В файл *output.txt* (см. рисунок 6.31) записываются не более N строк, каждая из которых содержит одну из N наибольших частот вхождения и соответствующие ей подпоследовательности. Строки располагаются в порядке убывания частот вхождения и имеют следующий вид:

частота подпоследовательности подпоследовательность ... подпоследовательность,

где частота — это количество вхождений указанных далее подпоследовательностей. Подпоследовательности в каждой строке должны располагаться в порядке убывания длины. При равной длине — в порядке убывания их числового значения. Если в последовательности встречается менее N различных частот, выходной файл будет содержать меньше N строк.

Пример.

input.txt

```
2
4
10
0101001001000100011110110000101001100111100001001001111001000000002
```

output.txt

```
23 00
15 10 01
12 100
11 001 000 11
10 010
8 0100
7 1001 0010
6 0000 111
5 1000 110 011
4 1100 0011 0001
```

Рис. 6.31

Указание. Входной файл очень большой. Следует провести обработку за один его просмотр.

Количество различных подпоследовательностей не так велико, как кажется. Их 8191 ($2^1+2^2+2^3+...+2^{12}=2^{13}-1=8191$) и это путь к реальному решению задачи. Введем массив D для хранения частоты встречаемости каждой подпоследовательности. Обрабатывая разряд за разрядом исходной последовательности, будем добавлять по единице к соответствующему элементу D. Ниже в таблице 6.10 приводятся результаты обработки первых 10 разрядов из примера, приведенного в условии задачи (единственное отличие — A считается равным 1). В первом столбце указана длина подпоследовательности, во втором — номер элемента массива D, в третьем — соответствующая подпоследовательность, далее номер обрабатываемого разряда и в последнем столбце — результат обработки.

Таблица 6.10

L	*N*		1	2	3	4	5	6	7	8	9	10	...	*D*
1	0	0	1		2		3	4		5	6			39
	1	1		1		2			3			4		26
2	2	00						1			2			23
	3	01		1		2			3			4		15
	4	10			1		2			3				15
	5	11												11
3	6	000												11
	7	001							1			2		11
	8	010			1		2			3				10
	9	011												5
	10	100						1			2			12
	11	101				1								3
	12	110												5
	13	111												6
4	14	0000												6
	15	0001												4
	16	0010								1				7
	17	0011												4
	18	0100						1			2			8
	19	0101				1								2
	20	0110												2
	21	0111												3
	22	1000												5
	23	1001							1			2		7
	24	1010					1							2
	25	1011												1
	26	1100												4
	27	1101												1
	28	1110												3
	29	1111												3
5	30	00000												
...	...	...												
12	4094	0...0												
...	...	...												

После того, как сформирован массив *D*, остается найти первые *N* максимальных значений и вывести затем соответствующие подпоследовательности. Остановимся на ключевых моментах обработки.

Разумно в массиве констант определить индексы для *D*, которые определяют первые последовательности заданной длины, что мы и сделаем.

```
Const Rt:Array[0..13] Of Integer=(0, 0, 2, 6, 14,
          30, 62, 126, 254, 510, 1022, 2046, 4094,
          8190);{*Rt[i] определяет, с какого
          номера начинаются последовательности
          длины i.*}
      MaxB=12;
      MaxN=21;
Var   D: Array[0..8190] Of LongInt;
      A, B, N: Integer;
```

При обработке очередного символа требуется вычислять последовательности, а лучше их номера, образуемые предыдущими символами и текущим символом из входной строки. Как это сделать?

Пусть в массиве *Sold* хранятся номера последних обработанных подпоследовательностей, например (0, 1, 1, 5, 5). Обработана следующая часть исходной последовательности — 0101, и мы обрабатываем символ '0' (Значение *B* равно 4, поэтому 5 элементов в массиве *Sold*). Что это значит (вспомните о массиве *Rt*)? В *Sold* хранятся текущие смещения, получаемые из последних символов входной строки, относительно первых констант заданной длины в массиве *D*. Так, первая единица означает, что последней была обработана последовательность '1', находящаяся на первом месте в своей группе последовательностей. Вторая единица — первая последовательность длины 2, т. е. '01' (начинаем отсчет в каждой группе с нулевого индекса). Первая пятерка — обрабатывалась последовательность длины 3 с пятого места, т. е. '101'. Последняя пятерка — обрабатывалась последовательность длины 4 с пятого места, т. е. '0101'.

Еще раз повторим. Подпоследовательности:

- длины 1 имеет 1-й номер, начиная с 0 (0+1=1 и соответствует последовательности '1' в массиве *D*);
- длины 2 — 1-й номер (2+1=3 и соответсвует последовательности '01' в массиве *D*);
- длины 3 — номер 5 (6+5=11 и соответствует последовательности '101' из таблицы);
- длины 4 — 5-й номер (14+5=19 и соответсвует последовательности '0101' из таблицы).

Переходим к вычислению новых номеров при обработке очередного символа '0' из входной строки:

- длины 1 – *0*×2+0, 0-й номер;
- длины 2 – *1*×2+0, подпоследовательность '10' имеет 2-й номер;
- длины 3 – *1*×2+0, подпоследовательность '010' имеет 2-й номер;
- длины 4 – *5**2+0, подпоследовательность '1010' имеет 10-й номер

и массив *Sold* имеет вид: (0, 0, 2, 2, 10). Обрабатываем следующий символ '0'.

- длины 1 – *0*×2+0, 0-й номер;
- длины 2 – *0*×2+0, 0-й номер;
- длины 3 – *2*×2+0, 4-й номер;
- длины 4 – *2*×2+0, 4-й номер.

Результат *Sold* — (0, 0, 0, 4, 4). Подпоследовательности '0' и '00' находятся на нулевых местах. Подпоследовательности '100' и '0100' на четвертых местах в своих группах. Проверьте. Откуда берем цифры выше по тексту, выделенные курсивом?

В таблице 6.11 приведено изменение значения элементов массива *Sold* при обработке первых 10 разрядов последовательности, приведенной в условии задачи.

Таблица 6.11

№ символа	Символ	*Sold*
1	0	0, 0, –1, –1, –1
2	1	0, 1, 1, –1, –1
3	0	0, 0, 2, 2, –1
4	1	0, 1, 1, 5, 5
5	0	0, 0, 2, 2, 10
6	0	0, 0, 0, 4, 4
7	1	0, 1, 1, 1, 9
8	0	0, 0, 2, 2, 2
9	0	0, 0, 0, 4, 4
10	1	0, 1, 1, 1, 9

```
Procedure Solve;
 Var:  nch: Char;
       Sold,Nw:Array[0..MaxB] Of Integer;
       i:Integer;
  Begin
    FillChar(Nw, SizeOf(Nw), 0);
    For i:=1 To MaxB Do Sold[i]:=-1;
    Sold[0]:=0;
```

```
    Read(nch);
    While nch<>'2' Do
      Begin{*Пока не прочитали последний символ.*}
        For i:=1 To B Do
          Begin
            If Sold[i-1]<>-1
              Then
                Begin
                  Nw[i]:=Sold[i-1] *2 +
                  Ord(nch)-Ord('0');
                  If (i>=A) Then Inc(D[Rt[i] + Nw[i]]);
                  {*Прибавляем единицу к соответствующему
                  элементу массива D.*}
                End
              Else Nw[i]:=-1;
          End;
        Sold:=Nw;{*Запоминаем вычисленную
              последовательность номеров.*}
        Read(nch);{*Читаем следующий символ.*}
      End;
  End;
```

Приведем еще одну вспомогательную процедуру. Мы работаем с номерами последовательностей, а на заключительном этапе требуется сама последовательность. Этот перевод осуществляется с помощью следующей простой процедуры.

```
Procedure SayW(num: Integer);{*По номеру
        подпоследовательности вычисляем ее вид.*}
  Var i, j: Integer;
      s: String;
   Begin
    i:=MaxB;
    While (i>0) And (Rt[i]>num) Do Dec(i);
         {*Определение длины.*}
    num:=num - Rt[i]; s:='';
    For j:=0 To i-1 Do
      Begin
        s:=Chr(num Mod 2 + Ord('0')) + s;
        num:= num Div 2;
      End;
    Write(' ', s);
   End;
```

23. Даны «цепочки» из нулей и единиц одинаковой (большой) длины. Над ними можно выполнять поразрядно операции *NOT*, *AND* и *OR*.

Таким образом: *OR*(*a*,*b*)=*NOT*(*AND*(*NOT*(*a*),*NOT*(*b*))).

Написать программу, которая вводит натуральное число N ($N \le 1995$), четыре цепочки *a, b, c*, и *d* длины N и определяет:

- можно ли последовательным применением перечисленных операций получить *d* из *a, b* и *c*;
- при удовлетворительном ответе строит нужную последовательность операций (достаточно одной последовательности).

Пример 1. Для N=6, *a*=011000, *b*=111010, *c*=010001 и *d*=100010 вывод программы может быть следующим:

- на первый вопрос — «Да»;
- на второй вопрос — *AND*(*NOT*(*a*),*OR*(*b*,*a*)).

Пример 2. Для N=6, *a*=0110000, *b*=111010, *c*=010001 и *d*=000010, ответ на первый вопрос отрицательный.

Указание. Если рассматривать разрядные «срезы» цепочек *a, b* и *c*, то различных всего восемь. Это наводит на мысль о том, что алгоритм обработки должен быть независим от длины цепочек, естественно, после их какой-то первоначальной обработки.

Поясним это утверждение на примерах.

Пусть:

a=11111111	01010101	10101010
b=10101010	00110011	00001111
c=01010101	00001111	11110000
2 «среза»	8 «срезов»	4 «среза»
d=01010101	10100101	11011010

Из этого факта следует, что цепочки при предварительной обработке необходимо «сжать». Это первое. Кроме того, очевидно, что если одному значению разрядных срезов (срезы по порядку различные) цепочек *a, b* и *c* соответствуют различные значения в соответствующих разрядах цепочки *d*, то на первый вопрос задачи имеется отрицательный ответ — «из одних и тех же данных невозможно получить два различных результата». Так, для третьего примера ответ — «Нет», первому и третьему разрядным срезам соответствуют различные значения в соответствующих разрядах цепочки *d*. Итак, предположим, что мы умеем сводить наши длинные цепочки (*a, b, c* и *d*) к цепочкам с длиной не более восьми разрядных срезов и решать первую часть задачи. Что из этого следует? Количество различных це-

почек длины не более 8 — 256. Это первое. И второе, какие бы операции над цепочками *a, b* и *c* мы ни выполняли, наш результат — это одна из 256 цепочек. Значит, необходимо организовать перебор вариантов не по различным последовательностям операций (*NOT, AND* и *OR*), а по результатам этих операций. А это уже только техника перебора и не более. Приведем часть структур данных, которые, на наш взгляд, соответствуют решаемой задаче.

```
Const Max=256;
Ch=Array[1..3] Of Char=('a','b','c');
Type Action=Record
          operation: 0..3;{*0 - нет операции; 1 -
                     NOT; 2 - OR; 3 - AND.*}
          op1,op2:Word;{*Номера операндов - индексы
                     элементов в массиве типа A.*}
           result:Byte{*Результат данной операции.*}
          End;
Var A: Array[0..Max] Of Action;
       cnt,oldcnt:Word;{*Счетчики числа записанных
                  элементов в массив A.*}
       ok:Boolean;
```

Первые четыре элемента массива *A* описывают исходные данные задачи после операции сжатия — значения полей *operation, op1, op2* равно нулю, *A*[0].*result=d, A*[1].*result=a, A*[2].*result=b* и *A*[3].*result=c*.

В качестве примера приведем одну из процедур — вывода результата. Входным параметром является номер (индекс) элемента массива *A* — первая операция, которая имеет результат, равный *d*.

```
Procedure Out(p:Word);
 Begin
  With A[p] Do
   Case operation Of
    0: Write(Ch[p]);
    1: Begin Write(' NOT ');Out(op1);End;
    2: Begin Write('(');Out(op1);Write(' OR ');
       Out(op2);Write(')');End;
    3: Begin Write('(');Out(op1);Write(' AND ');
       Out(op2);Write(')');End;
  End;
 End;
```

И основной алгоритм.

```
Begin
  <сжать исходные данные>;{*Результат - первые 4
элемента массива A и логическая переменная, в которой
фиксируется признак отсутствия решения.*}
  If <нет решения>
    Then <вывод сообщения>
    Else
      Begin
       cnt:=3;ok:=False;
       Repeat
        oldcnt:=cnt;
         <Перебор>;
       Until (oldcnt=cnt) Or ok;
      End;
End;
```

24. Заданы две строки A и B длины N ($1 \le N \le 100$), состоящие из символов 0 и 1 (двоичные слова)*. Допускается следующее преобразование строк. Любую подстроку A или B, содержащую четное число единиц, можно перевернуть, т. е. записать в обратном порядке. Например, в строке 11<u>0101</u>00 можно перевернуть подстроку, составленную из символов с 3-й по 6-ю позиции. Получится строка 11<u>1010</u>00.

Две строки считаются эквивалентными, если одну из них можно получить из другой с помощью описанных преобразований.

Написать программу, которая:

- определяет эквивалентность заданных строк A и B;
- если строки эквивалентны, предлагает один из возможных способов преобразования строки A в строку B.

Входные данные расположены в файле с именем *input.txt* (см. рисунок 6.32). В первой строке файла содержится число N. Следующие две строки содержат слова A и B соответственно. Слова записаны без ведущих и «хвостовых» пробелов.

Выходные данные. Результат работы программы выводится в файл с именем *output.txt* (см. рисунок 6.32). Если заданные слова не эквивалент-

Пример.

input.txt

```
9
100011100
001011001
```

output.txt

```
yes
6 9
3 8
1 5
```

Рис. 6.32

* Задачу предложил для одной из областных олимпиад по информатике Виталий Игоревич Беров.

ны, то выходной файл должен содержать только сообщение *no*. Если слова эквивалентны, то в первую строку выходного файла нужно вывести сообщение *yes*, в последующие — последовательность преобразований слова A в слово B. Каждое преобразование записывается в отдельной строке в виде пары чисел i, j (разделенных пробелом), означающей, что переворачивается подстрока, составленная из символов с номерами i, $i+1$, ..., j.

Указание. Основная идея решения задачи формулируется следующим образом. На множестве всех слов W следует выделить подмножество S ($S \subset W$), такое, что для любого слова $t \in W$ можно указать эквивалентное ему слово $r \in S$.

Алфавит и слова у нас заданы, преобразования тоже. Осталось определить множество S, свести каждое слово к представителю из множества S, и если представители совпадают, то слова эквивалентны.

Из каких слов состоит S? Слов вида 11...100...0100...0, причем как первая группа единиц, так и первая группа нулей, а также следующая единица (слово из одних 0) могут отсутствовать. Заметим, что вывод преобразований, в случае эквивалентности слов, чисто техническая деталь. Она требует дополнительной структуры данных (массива) для запоминания значений i и j при каждом перевертывании.

Другая схема решения (автор приводит рассуждения одного из восьмиклассников, решавших данную задачу). Как обычно, ребенок вначале рисует, рисует примеры слов и преобразований над ними. Попытаемся воспроизвести его рисунки. В таблице 6.12 столбцы с заголовком «?» — это уже выводы из рисунков.

Таблица 6.12

№	слово	число 1	?	?
1	1111000010	5	4	1
2	11111001	6	0	2
3	0000001000	1	6	3
4	10010001100	4	5	2
5	11000100100	4	5	2
6	11100100000	4	5	2
7	0100101101	5	3	2
8	1001001101	5	3	2
9	1100100101	5	3	2
10	1110010001	5	3	2
11	1111000100	5	3	2

Его вывод заключается в том, что *количество нулей* после четного числа единиц (4-й столбец) и после нечетного числа единиц (5-й столбец) в слове постоянно — не изменяются в процессе преобразования слов (строки таблицы с номерами 4-6 и 7-11). После этого он пишет небольшой фрагмент подсчета этих чисел (сразу после их ввода)

```
...
cnt:=0;
ch1:=0;ch2:=0;
For i:=1 To Length(S) Do
 If S[i]='0' Then
   If cnt Mod 2=0 Then Inc(ch1) Else Inc(ch2)
              Else Inc(cnt);
```

и выписывает результат — слово после его приведения. Нулевая позиция в слове считается при этом четной, например числа 1, 5 (5 нулей после четного числа единиц — нулевого), 4 (4 нуля после нечетного числа единиц) однозначно определяют слово 0000010000.

Обоснуем результат рассуждений. Инвариантом преобразования является знакочередующаяся сумма вида:

$$\sum(-1)^i \times a_i,$$

где i — номер единицы, а a_i — номер позиции i единицы.

Схема доказательства. Обозначим через s номер позиции в подстроке, соответствующий центру переворачиваемой подстроки. Пусть x — номер позиции до «переворота», y — номер этой позиции после операции «переворота». Очевидно, что $(x+y)/2=s$, т. е. $y=2s-x$, или $a_i \to 2s-a_i$.

Количество элементов в сумме четно, знаки чередуются и

$$\sum(-1)^i \times a_i = \sum(-1)^i \times (2s-a_i).$$

Поэтому после ввода слов следует подсчитать количество 1 в каждом из них и суммы указанного типа. Если они совпадают, то слова A и B эквивалентны.

Например, для строки 0101 сумма равна

$$(-1)^1 \times 2 + (-1)^2 \times 4 = 2,$$

после переворачивания — 1010, сумма —

$$(-1)^1 \times 1 + (-1)^2 \times 3 = 2.$$

Они совпадают, строки эквивалентны. Для примера (110100 и 110001) эти суммы равны −3 и −5, строки неэквивалентны.

Дополним задачу при ограничении $1 \le N \le 60$. Необходимо найти количество слов, эквивалентных заданному слову A.

Входные данные расположены в файле с именем *input.txt* (см. рисунок 6.33). В первой строке файла содержится число N. Следующая строка содержит слово A, записанное без ведущих и «хвостовых» пробелов.

Выходной файл с именем *output.txt* (см. рисунок 6.33) должен содержать искомое количество слов.

Задача на умение использовать динамические схемы подсчета, плюс — «длинная» арифметика.

Пример.

input.txt

9
101010011

output.txt

30

Рис. 6.33

Рассмотрим первую часть задачи. Мы подсчитали количество 0 после четного числа 1 в слове и после нечетного. Из вышеизложенного ясно, что перемещение 0 из той или другой групп по соответствующим позициям (после четного числа единиц или после нечетного) приводит к эквивалентным словам.

Приведем пример для пояснений. Пусть есть слово 101010010. Количество нулей после всех четных единиц в сумме равно 2, после нечетных — 3. Количество позиций, на которые можно переставлять нули из первой группы, равно 3, из второй — 2.

Выделим их — 101**0**10010**0**.

Переставляем «крупные» нули:

101100**00**, 101**00**1001, **00**1011001, **0**101**0**1001, **0**10110010 (подчеркнута та подстрока, которая переворачивается, но в данном случае это уже не принципиально).

Переставляем «мелкие» нули:

101**0**10010, 11**0**10010**0**, 100**1**0**1**0**0**, 10001**0**110.

Найдем для каждого случая количество эквивалентных слов, эти значения необходимо перемножить, результат — ответ задачи. Для нашего примера первое значение 6, второе — 4. Ответ — 24.

Возвращаясь к предыдущим рассуждениям, подсчитаем инварианты, например, для двух последних строк:

$$(-1)^1\times1+(-1)^2\times4+(-1)^3\times6+(-1)^4\times8=-1+4-6+8=5$$

и $(-1)^1\times1+(-1)^2\times5+(-)^3\times7+(-1)^4\times8=-1+5-7+8=5$.

Совпадают, строки эквивалентны.

Приведем текст процедуры, назовем ее *Find*, параметрами которой являются числа k и l — число позиций и число нулей, результат — число эквивалентных слов типа *LongInt*.

Пусть найдено решение для t позиций, т. е. определено число способов размещения для всех количеств нулей от 1 до l. При $(t+1)$-й позиции решение получается очевидным способом — в позиции с номером $t+1$ размещаем определенное количество нулей, в позициях с номерами от 1 до t — оставшиеся (они уже подсчитаны). Один из возможных вариантов реализации этой схемы приведен ниже, разумеется, это не единственный способ.

```
Function Find(k,l:Integer):LongInt;
 Var Q:Array[1..2,0..MaxN] Of LongInt;
       i,j:Integer;
 Begin
  FillChar(Q,SizeOf(Q),0);
  For j:=0 To l Do Q[1,j]:=1;
  For i:=1 To k-1 Do
    Begin
      Q[2,0]:=1;
      For j:=1 To l Do Q[2,j]:=Q[1,j]+Q[2,j-1];
      Q[1]:=Q[2];
    End;
  Find:=Q[1,l];
End;
```

25. Алгоритм Евклида для нахождения наибольшего общего делителя двух целых чисел хорошо известен и изучается в курсе основ информатики. Рассмотрим задачу повышенной сложности, идея решения которой основана на алгоритме Евклида. Даны натуральные числа p и q. Разработать программу определения последовательности натуральных чисел наибольшей длины N, такой, что значение суммы любых p идущих подряд элементов этой последовательности было бы положительно, а любых q — отрицательно*.

Разумно предположить, что последовательность имеет некоторую *регулярную структуру*, т. е. она не случайна. И эту регулярность следует найти.

Можно считать, что значение $N \geq max(p,q)$. Случай, когда $q=lp$ или $p=tq$ (одно из чисел делится нацело на другое) отсекается моментально. Действительно, пусть $q=lp$ и построена последовательность $a_1, a_2, \ldots, a_N$.

---

* Автор благодарит Александра Николаевича Семенова за помощь в разборе данной задачи.

Из равенства

$$\sum_{i=1}^{q} a_i = \sum_{i=1}^{p} a_i + \sum_{i=p+1}^{2p} a_i + \ldots + \sum_{i=(l-1)*p+1}^{l*p} a_i > 0$$

следует противоречие.

Сделаем два предположения: $q>p$ (иначе просто меняем знаки у элементов последовательности); p и q взаимно просты, т. е. НОД(p,q)=1. Затем результат обобщим.

Действия учащегося, ум которого не обременен техникой доказательств, выглядят достаточно просто — он пытается прорисовать последовательности. Попытаемся воспроизвести эти действия с помощью следующей таблицы 6.13 для произвольных значений p и q (на последний столбец таблицы пока не обращаем внимания).

Таблица 6.13

№	q	p	Схема Евклида	Последовательность	L	?
1	6	5	$6^-=1*5^++1^-$	----+-|---?	9	–2, 9
2	11	5	$11^-=2*5^++1^-$	----+----+-|---?	14	–3, 13
3	7	5	$7^-=1*5^++2^-$ $5^+=2*2^-+1^+$	+-+-++-|+-+?	10	–7, 5
4	12	5	$12^-=2*5^++2^-$ $5^+=2*2^-+1^+$	+-+-++-+-++-|+-+?	15	-10, 7
5	8	5	$8^-=1*5^++3^-$ $5^+=1*3^-+2^+$ $3^-=1*2^++1^-$	-+--+-+-|-+-?	11	–5, 8
6	13	5	$13^-=2*5^++3^-$ $5^+=1*3^-+2^+$ $3^-=1*2^++1^-$	-+--+-+--+-+-|-+-?	16	–7, 11
7	9	5	$9^-=1*5^++4^-$ $5^+=1*4^-+1^+$	+++-++++-|+++?	12	–11, 3
8	14	5	$14^-=2*5^++4^-$ $5^+=1*4^-+1^+$	+++-++++-++++-|+++?	17	–15, 4

Первая строка — q=6, p=5. Схема Евклида дает условный символ «–» (назовем его базовым). Конструируем из него последовательность. Сумма пяти элементов должна быть положительна. Добавляем другой символ — «+». Сумма шести элементов — отрицательна, поэтому добавляем «–». В таблице 6.13 эта часть отделена от остальной символом «|».

Что происходит дальше? Три базовых элемента (повторяют начало) не меняют структуры исследуемого нами объекта. В позицию, отмеченную «?», мы не можем записать ни «–», ни «+». Запись «–» на эту позицию делает последнюю сумму из пяти элементов отрицательной. Запись «+» — сумму из шести последних элементов положительной.

Чем отличается вторая строка таблицы 6.13? Последовательность «−−−−+» повторяется два раза (множитель 2 в схеме Евклида).

Третья строка таблицы более интересна. Из базового элемента мы конструируем последовательность из двух элементов (сумма их отрицательна). Из нее мы получаем последовательность длины пять и только затем длины семь, добавляя ранее полученную последовательность длины два.

Из этих же экспериментов определяется и максимальная длина $L - L=(p+q-2)$ при НОД$(p,q)=1$.

Действия математика (на наш взгляд) несколько отличаются от тех, что приведены. Он, быть может, тоже рисует последовательности, но в конечном итоге его результат выглядит иначе. Следует предположение о том, что $L>p+q-2$. Затем выполняются рассуждения по индукции, получается противоречие, и следствием является доказанный факт о том, что $L\le p+q-2$.

Итак, пусть мы получили последовательность (регулярную!) из «+» и «−». Требуется определить, какие натуральные числа соответствуют им (последний столбец таблицы). Должна быть *зависимость от значений p, q и соотношения количеств «+» и «−» на различных периодах последовательности*.

Обозначим через w_i число «+» на периоде t_i, тогда отношение числа «−» к числу «+» выражается $(t_i-w_i)/w_i$. Вернемся к таблице 6.13 и попытаемся определить, как связаны эти отношения на соседних периодах.

После некоторых усилий устанавливается следующий факт:

$$(t_i-w_i)/w_i<(t_{i+1}-w_{i+1})/w_{i+1}$$

при четном значении i и «>» при нечетном i.

Доказательство истинности этого факта математик проводит методом индукции. Что из него следует? Пусть $t_1=q$, $t_2=p$ и $i=1$.

Мы имеем неравенство

$$(q-w_1)/w_1>(p-w_2)/w_2 \quad (i \text{ — нечетно}).$$

Средневзвешенное чисел в правой и левой частях неравенства принадлежит интервалу, определяемому этими числами, т. е.

$$(q-w_1)/w_1>(q+p-w_1-w_2)/(w_1+w_2)>(p-w_2)/w_2.$$

Заменим «+» на число $x=p+q-w_1-w_2$ и «−» на число $y=-(w_1+w_2)$. Из последнего неравенства получаем, что сумма любых p чисел, идущих подряд, равна

$$w_2\times x+(p-w_2)\times y>0,$$

а сумма любых q чисел — равна

$w_1 \times x + (q - w_1) \times y < 0$.

После этого можно поставить точку в рассуждениях и перейти к написанию программы.

А случай НОД$(p,q) = d > 1$? Для значений p/d и q/d мы можем построить наш объект. Добавим между каждыми элементами построенной последовательности, а также справа и слева по $(d-1)$ нулей. Условия положительности и отрицательности сумм выполняются, а длина равна

$(p/d + q/d - 2) \times d + d - 1$, или $p + q - d - 1$.

7. Заметки о тестировании программ

При написании этой главы на память приходит один пример из собственной практики автора. После окончания университета автору пришлось разрабатывать программы (а это было начало 80-х годов прошедшего столетия) для специализированных вычислительных комплексов. В силу определенной живости натуры автор быстро стал ведущим разработчиком и отвечал за весь цикл разработки программ и их внедрение, будучи заместителем генерального конструктора по этим вопросам. Одну программную ошибку в комплексе, прошедшем все испытания и принятом заказчиком, автору пришлось искать не менее полугода.

Среди решаемых задач комплекса было построение связных областей (от любой точки области до любой точки можно попасть по точкам связной области) в четырехмерном пространстве параметров. Эти области изменялись во времени, одни точки области исчезали, другие добавлялись, области могли объединяться и наоборот. Вот такую динамичную картинку требовалось обрабатывать в реальном масштабе времени, причем поток входной информации (подлежащий обработке) был весьма значительным.

Периферийных устройств не было. Программы хранились в ПЗУ, и каждое изменение требовало его перепрограммирования. Для того, чтобы что-то просмотреть, необходимо было остановить соответствующий процессор и с помощью тумблеров набрать адрес регистра или ячейки, содержимое которых после этого отображалось с помощью диодов. Ошибка проявлялась нестабильно, только на определенных входных потоках, причем в одних случаях происходило зацикливание (не обязательно в одном месте), в других — бессмысленное «размножение» связных областей. Ошибка проявлялась достаточно редко, и это, а также то, что знание комплекса разработчиками было доскональное, позволило, видимо, пройти все испытания.

По характеру мигания светодиодов определялось место работы программы, устройства. Некоторые сбои устранялись простым ударом кулака по определенному месту соответствующей стойки и т. д. Ошибка оказалась достаточно простой.

В том случае, когда связные области в процессе изменений превращались в одиночные (или почти одиночные) точки и располагались в пространстве параметров примерно так, как черные клетки на шахматной доске, нехватало места в оперативной памяти для хранения их описания. Контроля на выход за пределы соответствующих структур данных не было (примерно так же, как выход индекса за пределы массива). Никто не предполагал такой ситуации, по всем предварительным расчетам ее не могло быть, однако... было получено огромное удовлетворение от проделанной работы.

Появление книги Э. Йодана [12] и др. было воспринято автором как должное, ибо нечто подобное он уже делал на практике, когда, например, в режиме «сухого плавания» (комплекса под рукой не было) искал описанную выше ошибку.

Приверженность технологии нисходящего проектирования программ осталась неизменной все эти годы. Дальнейшее развитие программирования, как науки, так и отрасли производства, только подтвердило правильность революционных идей, высказанных в свое время профессором Э. Дейкстрой.

7.1. О программировании

Программирование — теоретическая и практическая деятельность по обеспечению программного управления обработкой данных, включающая создание программ, а также выбор структуры и кодирования данных — классическое определение. Попробуем определить несколько иначе, более конструктивно, рассматривая то, чем занимается программист.

Есть задача или проблема. В первую очередь программист должен определить возможность ее решения, выбирая соответствующий метод. Затем разработать проект программы, состоящий из алгоритма на каком-либо из языков программирования, доказать правильность работы программы и предусмотреть возможность ее изменения, внесения изменений на этапе сопровождения. Таким образом, в укрупненном виде мы видим *три этапа*:

- до программирования,
- программирование,
- после программирования.

Есть еще и четвертый этап. После того, как все казалось бы сделано, ответить на вопрос — «а что если все не так?».

Только часть работы связана с выбором структур данных и кодированием — использованием языков программирования. Программирование есть, если так можно выразиться, инженер-

ная работа по конструированию некой целостной системы обработки данных. Отличие программы, например, от некоторой механической системы, в том, что число взаимодействующих частей в программе настолько велико, что не поддается никакому разумному объяснению и проверить работу этой программной системы, перебирать все возможные способы взаимодействия ее частей немыслимо даже на сверхбыстродействующих компьютерах в разумные сроки.

Итак, выделим это положение: *программа* (назовем так по традиции) — *сверхсложная система*. Где же выход при создании таких сверхсложных систем, при достижении их работоспособности? Видимо, только в технологиях, обеспечивающих на выходе качественный и надежный продукт, важнейшим элементом которых является умение тестировать программные решения.

7.2. Практические рекомендации

Общие положения.

Практические рекомендации по написанию программ основаны на технологии нисходящего проектирования.

О строгом математическом доказательстве правильности работы обычно не может быть и речи при достаточно сложной программе. Успешная трансляция не есть признак работающей программы, а правильность работы на некотором наборе тестов еще не означает, что программа всегда будет работать правильно, — следует помнить, что различных комбинаций входных данных бывает, как правило, бесконечно (или «практически» бесконечно) много. *Основная идея* — не пытаться программировать сразу. Пошаговая детализация автоматически заставляет формировать понятную структуру программы. При этом требуется отслеживать правильность детализации, создавая набор контрольных точек и просчитывая значения данных в них. Контрольные точки обычно проходятся при любых начальных данных (как точка сочленения в графе).

Выбор представления данных — один из фундаментальных аспектов разработки программы. Н. Вирт так определял критерии этого действия:

- *естественность*, структуры данных обязаны «вытекать» из специфики задачи;
- *привычность*, структуры данных выбираются так, чтобы программирование приближалось к утверждению «как говорю, так и пишу программу, ничего лишнего»;

- *оптимальность*, структуры данных выбираются так, чтобы была возможность построения эффективного алгоритма.

Например, решая задачу о построении каркаса минимального веса в графе, естественно выбрать список ребер, а не матрицу инциденций или матрицу смежности.

Примеры плохого программирования.

Приведем ряд примеров, которые не соответствуют технологии структурного написания программ.

1. Следует придерживаться правила о том, что любой фрагмент программного кода (алгоритма) должен иметь одну точку входа и одну точку выхода (пусть даже в ущерб эффективности использования памяти). Ниже приводится пример того, как небольшой фрагмент имеет две точки выхода.

```
While <условие 1> Do
  Begin
   ...
   If <условие 2> Then Exit;
   ...
  End;
```

Использование конструкции бесконечного цикла только усугубляет ситуацию.

```
While True Do
  Begin
   ...
   If <условие 2> Then Exit;
   ...
  End;
```

2. Искусственные приемы, как, например, продемонстрированный ниже случай «насильного» выхода из циклической конструкции типа *For*, приводят, при внесении изменений в программу, к многочисленным и трудно устранимым ошибкам.

```
For i:=1 To N Do
  Begin
   ...
   If <условие> Then i:=N+1;
   ...
  End;
```

3. Увлечение параметрами при работе с процедурами и функциями является «болезнью роста» при освоении технологии нисходящего проектирования программ.

```
Type BArray=Array[1..N] Of Byte;
...
Procedure Swap(Var B:BArray;i,k:Byte);
 Var x:Byte;
 Begin
   x:=B[i];B[i]:=B[k];B[k]:=x;
 End;
```

и вызов *Swap*(*B*,*i*,*k*) вместо следующего простого текста процедуры:

```
Procedure Swap(Var y,z:Byte);
 Var x:Byte;
 Begin
   x:=y;y:=z;z:=x;
 End;
```

и вызова *Swap*(*B*[*i*],*B*[*k*]).

4. Один профессиональный программист сказал автору в частной беседе о том, что для него критерием подготовленности выпускников высшего учебного заведения к работе является понимание (глубокое) рекурсии и знание динамических структур данных.

```
Procedure Generate;{*t - глобальная переменная*}
 Begin
  ...
  t:=t+1;
  Generate;
  t:=t-1;
  ...
 End;
```

Логичнее выглядел бы следующий фрагмент (вопрос об экономии места в стеке адресов возврата не ставится, оно есть и точка).

```
Procedure Rec(t:Byte);
 Begin
  ...
  Rec(t+1);
  ...
 End;
```

5. Очень трудно убедить школьников в том, что если переменная используется только в процедуре, то она и должна определяться в этой процедуре, пусть и с дублированием программного кода. Только через большое количество занятий, после многочисленных ошибок приходит понимание факта. Типичный пример такого написания программ приводится ниже.

```
Program
  Var i,j:Integer;
  Procedure A;
   Begin
    {Работа с переменными i, j};
   End;
  Procedure B;
   Begin
    {Работа с переменными i, j};
   End;
...
{**}
 Begin
  {Работа с переменными i, j}
 End.
```

Временна́я сложность(t).

Обычно на время решения задачи накладываются определенные ограничения. Размерность задачи опишем величиной N. Временная сложность алгоритма является линейной при $t \sim O(N)$, полиномиальной при $t \sim O(N^q)$, где q равно обычно 2 или 3, и экспоненциальной при $t \sim O(q^N)$. Эффективными являются алгоритмы первых двух типов.

Обычно подсказка о том, какой существует алгоритм решения, содержится в формулировке задачи (имеется в виду олимпиадный вариант). Если дано N, большее 50, то следует искать быстрый алгоритм, в противном случае сосредоточиться на эвристиках, сокращающих традиционный перебор с возвратом.

Ответим на вопрос: как оценить время работы программы в среде Турбо Паскаль?

Обычно используют тот факт, что к ячейке с абсолютным адресом $40:$6C (назовем ее *Timer*) один раз в 1/18.2 секунды аппаратно прибавляется единица (или один раз в каждые 55 миллисекунд).

Допустим, что программа должна работать 5 секунд (зафиксировали время в ячейке с именем *TimeTest*). В начале работы программы значение из $40:$6C запомнили в некоторой пере-

менной, например *TimeOld*. После этой подготовительной работы в расчетной части программы осталось проверять условие

$$Timer - TimeOld > 18.2 \times TimeTest.$$

Если оно выполняется, то время решения задачи истекло. Так как обычно перед завершением задачи требуется записать в выходной файл результат (хотя бы промежуточный, например наилучшее из найденных решений в переборной задаче), а для этого требуется время, то условие пишут с учетом этого факта

$$Timer - TimeOld > 18.2 \times (TimeTest - 0.5).$$

Ниже приведен текст фрагмента, в котором считается количество прибавления единицы к ячейке в зависимости от времени, выделенного на работу программы.

```
Program TimeCheck;
 Uses Crt;
 Const N=5;
 Var Timer:LongInt Absolute $40:$6C;
    TimeOld,TimeTest,L,i:LongInt;
    pp:Boolean;
 Begin
  ClrScr;
  For i:=1 To N Do
    Begin
      TimeOld:=Timer;{Запоминаем значение таймера.}
      pp:=True;
      TimeTest:=i;{*Время решения задачи.*}
      L:=0;
      While pp Do
        Begin {*Пока не истекло время,
            отведенное на решение задачи.*}
          Inc(L);{*Прибавляем единицу к счетчику.*}
          If Timer-TimeOld>18.2*(TimeTest{-0.5}) Then
            Begin
              pp:=False;
              WriteLn(L);
            End;
        End;
    End;
 End.
```

Приемы контроля правильности работы программы.

Сформулируем ряд приемов, позволяющих избежать простых ошибок в разрабатываемых программах.

1. Обычно формат входных данных задачи строго определен. Если в формулировке требуется контролировать корректность ввода данных, то следует выделить эту часть алгоритма в отдельную процедуру(ы). Однако в последние годы от требования проверки входных данных отказались (видимо, из-за простоты) — они считаются корректными. Несмотря на это, после ввода данных их следует сразу вывести и проверить, что позволит избежать ошибок этого типа (на них обычно не обращают внимания и только после значительных усилий возникает вопрос — «а правильно ли в программе вводятся данные»).

2. Иногда полезно отслеживать (не в пошаговом режиме) трассу работы определенной части программы. Так, при входе в процедуру можно выводить ее имя и значения входных параметров или выводить промежуточные данные в определенных точках программы. К этому же приему относится и введение дополнительных счетчиков, контролирующих прохождение различных ветвей алгоритма.

3. Синтаксические ошибки устраняются обычно моментально. Для поиска, если так можно выразиться, логических ошибок первого типа в среде Турбо Паскаль рекомендуется вести отладку с включенными директивами. Начало программы после нажатия *Ctrl+O+O* выглядит следующим образом:

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N-,O-,P-,Q-,R-,S+,T-,V+,X+}
{$M 16384,0,655360}.
```

Изменение, например, *R–* на *R+* позволяет контролировать выход индексов за пределы массива.

Логические ошибки следующего уровня сложности связаны обычно с тем, что задача была понята неверно, т. е. решена не та задача, которая была поставлена. В этом случае лучше всего «прогнать» решение на примере, просчитанном «вручную» (обычно такой пример приводится в формулировке задачи). Неверным может оказаться замысел алгоритма. Правильные решения получаются не для всех исходных данных. Это, вероятно, самый сложный тип ошибок и его рассмотрению посвящен следующий параграф данной главы.

О тестировании программ.

Тест является совокупностью исходных данных для проверки работоспособности программы. Одного теста обычно недостаточно для проверки каждой ветви программы, и говорят о

системе тестов. Кроме того, с помощью тестов проверяются количественные ограничения на исходные данные и временные ограничения. Исходя из этого, разумно формировать систему тестов, например, следующим образом:

первый тест должен быть максимально простым;

следующие тесты должны проверять вырожденные случаи (например, при поиске решений квадратного уравнения проверяются вырожденные случаи);

обязательно следует проверять граничные случаи (результат описан как тип *Integer*, а значение превосходит 32767);

предельные тесты проверяют решение при наибольших значениях входных параметров задачи, и, разумеется, требуется постоянный контроль над временем выполнения программы (при превышении ограничений по времени выбран не эффективный алгоритм решения).

Автоматизация проверки.

Обычно в задаче имена входного и выходного файлов определены. Постоянное изменение имени не очень рационально. Рассмотрим, как выходить из данной ситуации.

Во-первых, создается директория для работы с программой, и она является текущей в системе программирования Турбо Паскаль. Программа ищет входной файл в текущей директории и записывает в нее выходной файл.

Пусть есть следующая «варварская» программа — из файла считываются два числа, их сумма записывается в выходной файл. Кроме того, в программу вставлены не очень понятные операторы. Однако они написаны, и с ними проводится отладка.

```
Program add;
 Var a,b : Integer;
 Begin
  Assign(input,'input.txt'); Reset(input);
  ReadLn(a,b);
  Close(input);
  While a=10 Do;
  If a = 20 Then b := Round(a/0);
  Assign(output,'output.txt'); Rewrite(output);
  WriteLn(a+b);
  Close(output);
 End.
```

Создаем систему тестов (см. таблицу 7.1). В текущей директории записываем файлы с каждым тестом (имена *i.tst*, где *i* —

номер теста) и результирующие файлы (имена *i.ans*, где *i* — номер теста).

Таблица 7.1

Имя входного файла	Тест	Имя выходного файла	Результат
1.tst	2 5	*1.ans*	7
2.tst	45 45	*2.ans*	91
3.tst	3 5	*3.ans*	8
4.tst	10 10	*4.ans*	20
5.tst	20 0	*5.ans*	20

Переименование входных файлов в файл с именем *input.txt* и пятикратный запуск программы при очевидных результатах утомительно и не очень удобно. Требуется минимальная автоматизация процесса тестирования программ.

Ниже приведен текст программы. Следует набрать и сохранить его в файле *Checker.dpr* (текущей директории), а затем откомпилировать, используя среду *Delphi*. После этого запустите программу *Checker.exe*. Ее входными параметрами (они задаются в командной строке) являются:

имя исполняемого файла (рассмотренный выше пример следует откомпилировать в текущей директории, например, с именем *my.exe*);

количество тестов;

ограничение по времени (в секундах).

Вид командной строки —

Checker.exe my.exe 5 2 (5 тестов, 2 секунды на тест).

Результат очевиден. Итак, имея программу *Checker,* мы значительно упрощаем процесс отладки решений.

```
Program* Checker;
  {$apptype console} -{*Консольное приложение.*}
  uses Windows,SysUtils,Classes;
  var SInf       : StartUpInfo;{* Установки для
                   функции CreateProcess.*}
      PInf       : Process_Information;{*Информация
                   о созданном процессе.*}
      TickCount  : Longint;{* Количество миллисекунд.*}
```

---

* Программа разработана студентом 4-го курса факультета информатики Вятского государственного педагогического университета Московкиным Алексеем Александровичем.

```
    YourAnswer:  TStringList;{*Ответ тестируемой
                              программы.*}
    RightAnswer: TStringList;{*Правильный ответ.*}
    FileName: String;{*Имя тестируемой
                      программы.*}
    FeedBack: String;{*Результат тестирования.*}
    TestCount: Integer;{*Количество тестов.*}
    TimeLimit: Integer;{*Ограничение
                        по времени.*}
    IsTimeLimit: Boolean;{*Индикатор окончания
                          лимита времени.*}
    ExitCode: Cardinal;{*Код завершения
                    тестируемой программы.*}
    i: Integer;
begin
  FileName:= ParamStr(1);{*Первый параметр - имя
                                        файла.*}
  TestCoun:= StrToInt(ParamStr(2));{*Второй -
                           количество тестов.*}
  TimeLimit:= StrToInt(ParamStr(3));{*Третий -
                        ограничение по времени.*}
  for i:=1 to TestCount do
   begin
    RenameFile(IntToStr(i)+'.tst','input.txt');
       {*Переименовать входной тест в input.txt.*}
    FillChar(SInf,SizeOf(SInf),0);
    SInf.cb:= SizeOf(PInf);{*Заполнить
                            структуру.*}
    SInf.dwFlags:= STARTF_USESHOWWINDOW;
                   StartUpInfo
    SInf.wShowWindow:= sw_Hide;{*Сделать окно
                           процесса невидимым.*}
    CreateProcess(PChar(FileName),nil,nil,nil,
          false,{*Запуск тестируемой программы.*}
      CREATE_NEW_CONSOLE or NORMAL_PRIORITY_CLASS,
       nil,
      nil,SInf,PInf);
    TickCount  := GetTickCount;{*Запомнить время.*}
    IsTimeLimit := true;
    While ((TickCount+TimeLimit*1000) >=
      GetTickCount)and IsTimeLimit do
```

```
  if WaitForSingleObject(PInf.hProcess,0)<>
        WAIT_TIMEOUT then{*Ждать, пока программа
        закончит работу или истечет тайм-аут.*}
     IsTimeLimit := false;
  if IsTimeLimit then
    begin {*Если истек тайм-аут,
           закончить работу программы.*}
      TerminateProcess(PInf.hProcess,0);
      FeedBack := 'Time limit exceeded!';
        {*Превышение лимита времени.*}
    end
                  else
    begin
      GetExitCodeProcess(PInf.hProcess,ExitCode);
           {*Получить код завершения программы.*}
      if ExitCode<>0
        then FeedBack := 'Run-time error!'
          {*Если код завершения не равен нулю, то
          ошибка времени выполнения.*}
        else
         begin
          YourAnswer  := TStringList.Create;
          RightAnswer := TStringList.Create;
          YourAnswer.LoadFromFile('output.txt');
           {*Загрузить файл с ответами тестируемой
            программы.*}
          RightAnswer.LoadFromFile(IntToStr(i)+'.ans');
           {*Загрузить файл с правильными
           ответами.*}
          if YourAnswer.Text<>RightAnswer.Text
            then
              Feedback := 'Wrong answer ...'  {*Если
              они не совпадают, то тест не пройден,*}
            else
             FeedBack := 'Ok.';{*иначе все в порядке.*}
          YourAnswer.Free;
          RightAnswer.Free;
         end;
    end;
  RenameFile('input.txt',IntToStr(i)+'.tst');
       {*Переименовать входной тест.*}
  Writeln('Test '+IntToStr(i)+':  '+FeedBack);
       {*Выдать результат теста.*}
end;
```

```
  DeleteFile('output.txt');{*Удалить ненужный
                            файл.*}
  writeln('Completed ...');{*Выдать сообщение об
                            окончании.*}
  Readln;
end.
```

7.3. Тестирование программы решения задачи (на примере)

Рассмотрим следующую задачу.

Даны два квадратных клеточных поля (см. рисунок 7.1). Размер клетки 1×1. Клетки поля закрашены в белый или черный цвет. Совпадающая часть квадратов — это множество клеток, имеющих одинаковый цвет. Она разбивается на какое-то количество связных областей. Найти площадь наибольшей связной области совпадающей части квадратов.

Пример

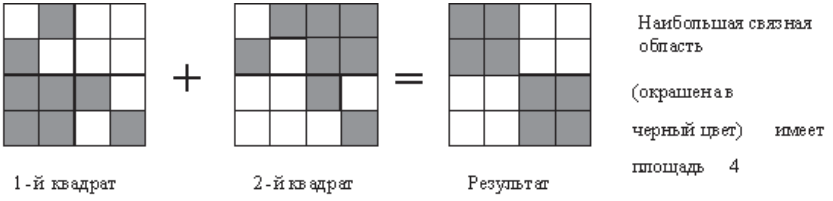


Рис. 7.1

Описание квадратов хранится в текстовых файлах (см. рисунок 7.2). Символ «1» во входных файлах соответствует черной клетке, символ «0» — белой. В выходном файле должно быть одно целое число, равное площади наибольшей связной области общей части.

Ограничения. Размер поля N в интервале от 2 до 1000. Время работы — не более 5 секунд.

Анализ и решение задачи. В идейном плане задача очень простая. Ее суть сводится к следующему:

```
If <Клетка[i,j] 1-го квадрата>=Клетке[i,j]
     2 квадрата
 Then <Клетке[i,j] результата>: = 1
 Else <Клетка[i,j] результата>: = 0,
```

Пример.

Photo1.txt

```
4
0100
1000
1110
1101
```

Photo2.txt

```
4
0111
1011
0010
0001
```

Result.txt

Рис. 7.2

а затем найти связную область с наибольшей площадью.

Проработав этот учебник до данной страницы, программное решение представленное ниже, пишется за считанные минуты.

Структуры данных.

```
Const
 Size = 100;{*Размер квадрата.*}
 Dx: Array[1..4] Of Integer = (0, -1, 0, 1);
       {*Вспомогательные массивы Dx, Dy
        используются в волновом алгоритме обхода
        связной области.*}
 Dy: Array[1..4] Of Integer = (-1, 0, 1, 0);
Type
 PhotoMatch = Array [1..Size, 1..Size] Of Byte;
       {*Для хранения результирующего квадрата.*}
 Helper = Array [1..Size*Size] Of Integer;
       {*Вспомогательный тип для хранения очереди
        при обходе в ширину.*}
Var
 PhotoM : PhotoMatch;{*Результирующий квадрат.*}
 Ox,Oy: Helper;
 Result, TempRes,N: Integer;
```

Напомним алгоритм волнового алгоритма (п. 3.2).

```
Function Get(Const k,p : Byte): Integer;{*Обход
         совпавшей части квадратов.*}
 Var i, dn, nx, ny, up : Integer;
 Begin
  Ox[1] := k; Oy[1] := p; dn := 1 ; up := 1;
        {*Заносим координаты первой клетки
        в очередь.*}
  PhotoM [k,p]:=0;
  Repeat
   For i:=1 To 4 Do
     Begin
      nx := Ox [dn] + dx [i]; ny := Oy [dn] + dy [i];
      If (nx>0) And (ny>0) And (nx<=N) And (ny<=N)
        And (PhotoM[nx,ny]=1)
       Then
        Begin{*Заносим в очередь.*}
          PhotoM[nx, ny] := 0; Inc(up); Ox [up]:= nx;
          Oy [up]:= ny;
        End;
     End;
```

```
   Inc(dn);{*Переходим к очередному элементу
           очереди.*}
  Until dn>up;{*Пока очередь не пуста.*}
  Get := up;{*Количество элементов в очереди равно
           площади общей части.*}
End;
```

Процедура ввода данных.

```
Procedure Init;
 Var photo1, photo2: Text;
     char1, char2 : Char;
     x,y : Integer;
 Begin
  Assign (photo1, 'photo1.txt'); Reset(photo1);
        {*Входные файлы.*}
  Assign (photo2, 'photo2.txt'); Reset(photo2);
  ReadLn(photo1,N);ReadLn(photo2,N);
  For y := 1 To N Do
    Begin
      For x := 1 To N Do
        Begin
          Read(photo1, char1); Read(photo2, char2);
          PhotoM [x,y]:=Ord(char1=char2);{*Если
             "закраска" совпадает, то …*}
        End;
      ReadLn(photo1); ReadLn(photo2);
    End;
 End;
```

Главная процедура.

```
Procedure Solve;
 Var i, j,t :Integer;
 Begin
  Result:=0;
  For i:= 1 To N Do
   For j:= 1 To N Do
    If (PhotoM[i,j]=1) Then
      Begin
        t:=Get(i,j);
        If t>Result Then Result:=t;{*Площадь
           наибольшей связной области.*}
      End;
 End;
```

Вывод результата.

```
Procedure WriteAnswer;
 Var f: Text;
 Begin
  Assign (f , 'result.txt'); Rewrite(f);
  Write (f, Result); Close(f);
 End;
```

Основная программа.

```
Begin
  Init;
  Solve;
  WriteAnswer;
End.
```

Итак, часть тестов, вне всякого сомнения, проходит, и есть база для проведения экспериментов. Изменение значения константы *Size* до требований задачи (даже до значения 200) приводит к сообщению *Error 22: Structure too large*. Не хватает оперативной памяти. Напомним, что в среде Турбо Паскаль для задачи выделяется порядка 64 Кбайт*.

Размерность задачи такова, что результирующее клеточное поле хранить в оперативной памяти можно только с использованием динамических структур, битовых записей, или еще каким-нибудь оригинальным способом. А требуется ли хранить? Возможна построчная обработка поля.

Пусть обработана строка. При обработке следующей строки возможны ситуации (см. рисунок 7.3):

область 1 продолжается в следующей строке;

область 2 закончилась; области 3 и 4 «сливаются» в одну;

область 5 только начинается.

Рис. 7.3

---

* В настоящее время происходит отказ от среды Турбо Паскаль на Международных олимпиадах по программированию. В новых средах таких жестких ограничений по оперативной памяти, как правило, нет. Автор считает, что ограничения — один из «двигателей» развития. Вспомним 1-й и 2-й этапы развития технологий программирования. Ограничения по ресурсам ЭВМ являлись мощнейшим рычагом создания эффективных алгоритмов и их исследования.

Единственная небольшая сложность — правильно отслеживать изменение значений площади при слиянии областей.

Основные процедуры этого варианта программы приводятся ниже по тексту.

```
Const Size = 1000;{*Размер поля.*}
Type
  Answer = Array [0..Size] Of LongInt;{*Массив
       ответов (ячейки с площадями).*}
  PhotoLine = Array [0..Size] Of Integer;{*Для
       обработки строки поля.*}
Var
  OldLine, NewLine : PhotoLine;{*OldLine -
       предыдущая, NewLine - новая строки.*}
  wrInd : Integer;{*Рабочая ячейка, для поиска
       свободного номера связной области.*}
  Result : Integer;{*Результат.*}
  TempRes : Answer;
  file1, file2 : Text;
```

```
Procedure Init;
Begin
  FillChar( OldLine, SizeOf(OldLine), 0);
  FillChar( TempRes, SizeOf(TempRes), 0);
  wrInd := 1;
  Result:=0;
End;
```

```
Procedure Change(DelInd, NewInd: Integer);
     {*Заменяем номера устаревших связных областей
      на новое значение.*}
 Var i : Integer;
 Begin
  For i := 1 To Size Do
    Begin{*Выполняется при
       слиянии двух связных областей.*}
      If NewLine[i]=DelInd Then NewLine[i]:= NewInd;
      If OldLine[i]=DelInd Then OldLine[i]:= NewInd;
    End;
 End;

Procedure CompareLines;{*Сравнения предыдущей и
       новой строк. Находим, какие связные области
       сливаются или дают "потомство".*}
```

```
 Var i :Integer;
     char1, char2 :Char;
 Begin
  FillChar( NewLine, SizeOf(NewLine), 0);
  For i := 1 To Size Do
Begin
   Read(file1, char1); Read(file2, char2);
   If char1 = char2 Then
     Begin {*Правило одинаковой
         "закрашенности".*}
      If (NewLine[i-1] = 0) Or (i = 1)
        Then
          Begin
            {*Новая связная область.*}
           While TempRes[wrInd]<>0 Do
            If wrInd<Size
              Then Inc( wrInd )
              Else wrInd:= 1;
           NewLine[i] := wrInd
          End
        Else NewLine[i] := NewLine[i-1];{*Простое
           дополнение к связной области.*}
      If (OldLine[i]<>0) And (NewLine[i]<>OldLine[i])
           Then
            Begin {*Слияние или наследование.*}
             TempRes[ OldLine[i] ] := TempRes
             [ OldLine[i] ] + TempRes[ NewLine[i] ];
             {*Сложение значений площадей.*}
              If TempRes[ NewLine[i]]<>0
                Then
                  Begin
                   {*Слияние.*}
                   TempRes[ NewLine[i] ] := 0;
                   {*Уничтожение устаревшего описания
                   связной области.*}
                   Change(NewLine[i], OldLine[i]);
                  End
                Else NewLine [i] := OldLine [i];
                 {*Наследование.*}
            End;
       Inc( TempRes[ NewLine[i] ] );
      End
     Else NewLine[i] := 0;
```

```
      End;
      ReadLn(file1); ReadLn(file2);
    End;

Procedure Solve;
 Var i, j : Integer;
 Begin
  Assign(file1, 'photoA.txt'); Reset(file1);
  Assign(file2, 'photoB.txt'); Reset(file2);
  For i := 1 To Size Do
    Begin
      CompareLines;{*Сравниваем строки.*}
      For j := 1 To Size Do
       If (TempRes[j]<>0) And (TempRes[j] > Result)
          Then Result :=TempRes[j];
      OldLine := NewLine;{*Новую строку считаем
                          старой.*}
    End;
  Close(file1);Close(file2);
End;
```

Продолжим тестирование решения. Вопросы с оперативной памятью решены. Остались временны́е ограничения — 5 секунд на работу программы с одним тестом. С этой целью необходимо разработать достаточно полную систему тестов. Один тест максимального размера еще не говорит о том, что выполняются требования задачи. Один из файлов (для простоты) можно заполнить единицами (первая вспомогательная программа). А затем? Напрашивается создание файлов максимального размера по принципам, приведенным на рисунке 7.4, и, конечно, файла со случайными значениями, а это еще четыре вспомогательные программы (не ручным же способом заполнять файлы).

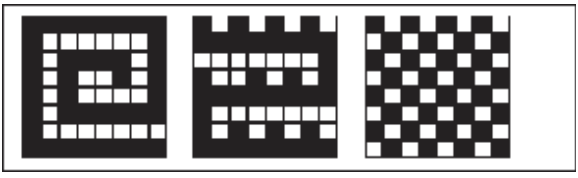


Рис. 7.4

Этим, вероятно, не исчерпываются все возможности, но ограничимся приведенными тестами — их достаточно для на-

шего изложения. Затем необходимо организовать учет времени решения задачи одним из способов, описанным выше (лучше, естественно, вторым). Что выясняется? Оказывается, только на одном из тестов временное ограничение не выполняется, а именно на тесте со структурой — «зубья» (см. рисунок 7.5).

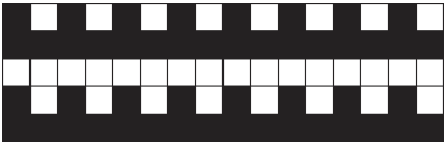


Рис. 7.5

Почему именно на этом тесте? Вернемся к решению. Ответ просматривается — частота слияния связных областей на тесте значительно возрастает, что вызывает работу процедуры *Change*, т. е. лишний цикл (подсчитайте — сколько раз выполняется процедура *Change* для области 1000×1000, заполненной по принципу «зубьев»).

Можно ли без принципиальных изменений на идейном уровне и в программной реализации добиться выполнения требований задачи? Оказывается, да! Используем идею косвенной адресации. Исключим, точнее изменим, процедуру *Change* из решения. В процедуре просматривался весь массив с целью поиска существующих «родственников». Изменим смысловую нагрузку элемента массива. Она будет содержать:

- или данные о площади;
- или ссылку (информацию) на ту ячейку, в которой записано значение площади при слиянии связных областей;
- или ссылку на ссылку при многократном слиянии связных областей.

Как их отличить друг от друга? Стандартный прием — отрицательное значение является признаком косвенной адресации. Исключение многократного просмотра длинных цепочек (при слиянии большого количества связных областей) осуществляется одноразовым прохождением по цепочке, после которого во всех ячейках указывается адрес конечной ячейки со значением площади. Еще один момент рассматриваемой идеи требует упоминания. В предыдущем варианте признаком свободного номера связной области являлось нулевое значение элемента в массиве *TempRes*. В данном варианте этот массив используется для косвенной адресации к ячейкам со значением площади, поэто-

му требуется введение дополнительного массива с элементами логического типа для этой цели, назовем его *Used*.

```
Type  UsedAn = Array [1..Size] Of Boolean;{*Тип для
      хранения информации об использованных номерах
      связных областей.*}
Var Used:UsedAn;
```

При этом предположении поиск свободного номера связной области выглядит следующим образом.

```
Procedure NextWrInd;{*Массив Used необходим,
    значение Used[i], равное False, говорит о том,
    что значение i - свободный номер связной
    области.*}
 Begin
  While Used[wrInd] Do
    If wrInd<Size Then Inc( wrInd ) Else wrInd := 1;
  TempRes [ wrInd ] := 0;
End;
```

Ключевая функция поиска номера ячейки, указывающей на значение площади. При выходе из рекурсии изменяем значения ссылок.

```
Function FindLast( ind : Integer ):Integer;{*Ищет
    итоговый номер, указывающий на значение
    площади.*}
 Begin
  If TempRes [ ind ] > 0
    Then FindLast := ind
    Else
      Begin
        TempRes [ ind ] := -FindLast ( Abs ( TempRes
                                          [ind] ) );
        FindLast := Abs ( TempRes [ ind ] );
      End;
 End;
```

Процедура сравнения соседних строк претерпит незначительные изменения. Приведем ее для сравнения с предыдущей.

```
 Procedure CompareLines;
  Var i : Integer;
     char1, char2 : Char;
  Begin
```

```
  FillChar( NewLine, SizeOf(NewLine), 0);
  For i := 1 To Size Do
   Begin
    Read(file1, char1); Read(file2, char2);
     If charA = charB Then
      Begin
       If (NewLine[i-1] = 0) Or (i = 1)
        Then
         Begin
          NextWrInd;{*Новый вариант поиска
          свободного номера связной области.*}
          newLine[i] := wrInd;
         End
        Else NewLine[i] := NewLine[i-1];
     If (OldLine[i]<>0) And
       (NewLine[i]<>OldLine[i])
         Then
          If TempRes [ OldLine[i] ] > 0
           Then
            Begin
             {*Простое слияние.*}
             TempRes[OldLine[i]]:=
             TempRes[OldLine[i]]+
             TempRes[NewLine[i]]; {*Суммируем
             площади.*}
             TempRes[ NewLine [i] ]:= - OldLine [i];
             {*Преобразуем ячейку со значением
              площади в ячейку ссылочного типа.*}
             NewLine[i] := OldLine [i]
            End
           Else NewLine [i] := FindLast ( OldLine
              [i] ); {*При слиянии со связной
              областью, которая при данном
              значении i помечена ячейкой
              ссылочного типа, находим ячейку со
              значением площади этой связной
              области. *}
      Inc( TempRes[ NewLine[i] ] );
      Used [NewLine [i] ] := True ;
     End;
    End;
   ReadLn(file1);ReadLn(file2);
End;
```

И наконец, новый вариант процедуры *Solve*.

```
Procedure Solve;
 Var i:Integer;
 Begin
  InitTemp;
  Assign(fileA, 'photoA.txt'); Reset(fileA);
  Assign(fileB, 'photoB.txt'); Reset(fileB);
  For i := 1 To Size Do
   Begin
    CompareLines;
    FillChar ( used , SizeOf(used), False);
         {*Считаем все номера свободными.*}
    For j := 1 To Size Do
      Begin
       If TempRes [ NewLine [j] ] < 0 Then
        NewLine [j] := FindLast ( NewLine [j] );{*Этим
           действием мы сокращаем количество
           переадресаций по ссылочным ячейкам при
           поиске ячейки с площадью, значительно
           ускоряя тем самым обработку следующей
           строки при слиянии связных областей.*}
       If (TempRes [ NewLine [j] ] > 0) And Not Used
          [NewLine [j]] Then
         Begin
          If TempRes [ NewLine [j] ] > Res Then
             Res := TempRes [ NewLine [j] ];
          Used [ NewLine [j] ] := True;
         End;
      End;
    OldLine := NewLine;
   End;
  Close(fileA);Close(fileB);
 End;
```

Для иллюстрации алгоритма, а именно изменения значений в массивах *NewLine* и *TempRes* (см. таблицу 7.2), рассмотрим простой пример, первые две строки которого изображены на рисунке 7.6.

Рис. 7.6

Таблица 7.2

i	*NewLine*	*TempRes*	Примечание
	1020304050	1111100000	После обработки 1-й строки сформировано пять связных областей, площадь каждой из них равна 1
			Обрабатываем 2-ю строку. В 1-м столбце таблицы указывается номер клетки строки
1	6000000000 1000000000	11111–10000 21111–10000	Указывается последовательное изменение значений в массивах при работе логики
2	1100000000	31111–10000	Простое добавление к связной области
3	1120000000	34111 –10000 –24111–10000 –25111–10000	«Склейка» связных областей. Площадь записана во второй ячейке. Первая переходит в разряд ссылочных
4	1122000000	–26111–10000	Простое добавление к связной области
5	1122330000	–2–3811–10000	«Склейка» с 3-й связной областью из предыдущей строки. Приводится окончательный вид массива *TempRes*
...	...	...	...
9	1122334450	–2–3–4–514–10000	«Склейка» с 5-й связной областью из предыдущей строки
10	1122334455	–2–3–4–515–10000	Работа процедуры *CompareLines* окончена
	5555555555	–5–5–5–515–10000	Готовим данные для следующего вызова *CompareLines*. Цепочка ссылок сокращена до одной – на ячейку с площадью

Этот вариант программы «разбирается» с тестом типа «зубьев» размером 1000×1000 за 3 секунды. Можно ли продолжить совершенствование программы решения задачи? Безусловно, ибо нет предела совершенству. Мы не использовали динамические структуры данных. А может быть, есть вариант не построчного сравнения клеточных полей. Да даже написание программ, генерирующих входные файлы различных типов, интереснейшая задача.

Библиографический указатель

1. Алексеев А. В. Олимпиады школьников по информатике. Задачи и решения. — Красноярск: Красноярское книжное издательство, 1995.

2. Андреева Е. В., Фалина И. Н. Информатика: Системы счисления и компьютерная арифметика. — М.: Лаборатория Базовых Знаний, 1999.

3. Бабушкина И. А., Бушмелева Н. А., Окулов С. М., Черных С. Ю. Конспекты занятий по информатике (практикум по Паскалю). — Киров: Изд-во ВятГПУ, 1997.

4. Бадин Н. М., Волченков С. Г., Дашниц Н. Л., Корнилов П. А. Ярославские олимпиады по информатике. — Ярославль: Изд-во ЯрГУ, 1995.

5. Беров В. И., Лапунов А. В., Матюхин В. А., Пономарев А. А. Особенности национальных задач по информатике. — Киров: Триада-С, 2000.

6. Брудно А. П., Каплан Л. И. Московские олимпиады по программированию. — М.: Наука, 1990.

7. Виленкин Н. Я. Комбинаторика. — М.: Наука, 1969.

8. Вирт Н. Алгоритм+структуры данных=Программы. — М.: Наука, 1989.

9. Гусев В. А., Мордкович А. Г. Математика: Справочные материалы. — М.: Просвещение, 1990.

10. Дагене В. А., Григас Г. К., Аугутис К. Ф. 100 задач по программированию. — М.: Просвещение, 1993.

11. Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И. Лекции по теории графов. — М.: Наука, 1990.

12. Йодан Э. Структурное проектирование и конструирование программ. — М.: Мир, 1979.

13. Касаткин В. Н. Информация. Алгоритмы. ЭВМ. — М.: Просвещение, 1991.

14. Касьянов В. Н., Сабельфельд В. К. Сборник заданий по практикуму на ЭВМ. — М.: Наука, 1986.

15. Кирюхин В. М., Лапунов А. В., Окулов С. М. Задачи по информатике. Международные олимпиады 1989-1996. — М.: «ABF», 1996.

16. Кристофидес Н. Теория графов. Алгоритмический подход. — М.: Мир, 1978.

17. Лапунов А. В., Окулов С. М. Задачи международных олимпиад по информатике. — Киров, Изд-во ВятГПУ, 1993.

18. Липский В. Комбинаторика для программистов. — М.: Мир, 1988.

19. Окулов С. М., Пестов А. А. 100 задач по информатике. — Киров, Изд-во ВятГПУ, 2000.

20. Окулов С. М., Пестов А. А., Пестов О. А. Информатика в задачах. — Киров, Изд-во ВятГПУ, 1998.

21. Окулов С. М. Конспекты занятий по информатике (алгоритмы на графах): Учебное пособие. — Киров, Изд-во ВятГПУ, 1996.

22. Окулов С. М. Задачи кировских олимпиад по информатике. — Киров, Изд-во ВятГПУ, 1993.

23. Окулов С. М. Основы программирования. — М.: Лаборатория Базовых Знаний, 2001.

24. Препарата Ф., Шеймос М. Вычислительная геометрия: введение. — М.: Мир, 1989.

25. Савельев Л. Я. Комбинаторика и вероятность. — Новосибирск: Наука, 1975.

26. Усов Б. Б. Комбинаторные задачи// Информатика. 2000. № 39.

Учебное электронное издание

Серия: «Развитие интеллекта школьников»

Окулов Станислав Михайлович

ПРОГРАММИРОВАНИЕ В АЛГОРИТМАХ

Редактор *О. А. Полежаева*
Художественный редактор *Н. А. Новак*
Технический редактор *Е. В. Денюкова*
Компьютерная верстка: *Л. В. Катуркина*

Подписано 27.11.13. Формат 60×90/16.
Усл. печ. л. 23,94.

Издательство «БИНОМ. Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: binom@Lbz.ru
http://www.Lbz.ru, http://e-umk.Lbz.ru, http://metodist.Lbz.ru

Минимальные системные требования определяются соответствующими требованиями программы Adobe Reader версии не ниже 10-й для операционных систем Windows, Android, iOS, Windows Phone и BlackBerry

РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

Очередная книга из серии «Развитие интеллекта школьников» снова вводит вас в мир программирования.
Внимательно изучайте эту книгу – и вам откроются секреты популярных алгоритмов!
Вы познакомитесь с комбинаторными алгоритмами, перебором, алгоритмами на графах, алгоритмами вычислительной геометрии, а также получите навыки решения олимпиадных задач

Станислав Михайлович ОКУЛОВ

Декан факультета информатики Вятского государственного гуманитарного университета, кандидат технических наук, доктор педагогических наук, профессор, почетный работник высшего профессионального образования РФ. Автор 9 изобретений по элементам ассоциативных вычислительных структур и автор (соавтор) 15 книг по информатике для школьников и студентов.

Книги автора, вышедшие в серии «Развитие интеллекта школьников»:

«Основы программирования»
«Ханойские башни»
«Абстрактные типы данных»
«Алгоритмы обработки строк»
«Динамическое программирование»

Область интересов: развитие интеллектуальных способностей школьника при активном изучении информатики, методика преподавания информатики в школе и вузе.

С 1993 по 2003 год деятельность в вузе совмещал с работой учителя информатики. За это время его ученики отмечены 33 дипломами (1-й и 2-й степени) на российских олимпиадах школьников по информатике; трое из них представляли Россию на международных олимпиадах.